TRANSIMS Version 1.1

Volume Five

Software: Interface Functions and Data Structures

LA-UR-00-1755

Disclaimer

These archived, draft documents describe TRANSIMS, Version 1.1, covered by the university research license. However, note that the documentation may be incomplete in some areas because of the ongoing TRANSIMS development. More recent documentation (for example, Version 2.0) may provide additional updated descriptions for Version 1.1, but also covers code changes beyond Version 1.1.

1. SYNTHETIC POPULATION

1.1 Interface Functions

The synthetic population subsystem has C structures and utility functions that are used to read and write synthetic population data from TRANSIMS synthetic population files.

1.1.1 moreSyntheticHH

Signature `int moreSyntheticHH(FILE* const fp`

Description Boolean function used to control iteration through the synthetic population file.

Argument `fp` – `FILE*` for the synthetic population file that must be open for reading.

Return Value 1 if not at end of synthetic population file.
0 if EOF has been reached.

1.1.2 getNextSyntheticHH

Signature `const SyntheticHHData* **getNextSyntheticHH**(FILE* const fp)`

Description Reads a synthetic household from the synthetic population file. Parses and converts the values from the file and stores them in the static `SyntheticHHData` structure.

Argument `fp` – `FILE*` for the synthetic population file that must be open for reading.–

Return Value The address of a static `SyntheticHHData` structure containing the data read from the file. Returns `NULL` on error.

1.1.3 writeSyntheticPopHeader

Signature
```
int writeSyntheticPopHeader(FILE* const fp,
char* hh_header, char* p_header)
```

Description Writes the header lines in the synthetic population file.
The format of the line is:

```
<text>: <demog1> <demog2> … <demogN>
```

Example:

```
Household Demographics: PUMSHH R18UNDR RWRKR89
```

```
RHHINC Person Demographics: AGE RELAT1 SEX
WORK89
```

Argument `fp` – pointer to synthetic population file that must be open for writing with the file pointer positioned at the beginning of the file.
`hh_header` – string containing the household header information.
`p_header` – string containing the person header information.

Return Value 1 on success.
0 on error.

1.1.4 CreatePopulationIndex

Signature `int` **`CreatePopulationIndex`**`(const char *popFileName)`

Description Creates an index to a TRANSIMS synthetic population file sorted by home location as the primary key, and household ID as the secondary key.

Argument `popFileName` – the name of the TRANSIMS synthetic population file.

Return Value 1 on success.
0 on error.

1.1.5 getSyntheticHHFromString

Signature `const SyntheticHHData*` **`getSyntheticHHFromString`**`(const char *data)`

Description Reads and parses the data for a `SyntheticHHData` record from a data string. The data string may not be null terminated.

Argument `data` – the data character string

Return Value The address of a static `SyntheticHHData` structure containing the data read from the string.
`NULL` on error.

1.1.6 writeSyntheticHH

Signature `int` **`writeSyntheticHH`**`(FILE* const fp,`
`const SyntheticHHData* hh)`

Description Writes the given `SyntheticHHData` into the given synthetic population file.

Argument `fp` – `FILE*` to the synthetic population file that must be open for writing.
`data` – address of a `SyntheticHHData` structure containing the data to be written.

Return Value 1 on success.
0 on error.

1.1.7 getSyntheticHouseholdFromIndex

Signature
```
const SyntheticHHData*
getSyntheticHouseholdFromIndex(BTree *index,
BTreeEntry *key)
```

Description Get the data for the household from an indexed file.

Argument `index – BTree *` – A BTree index for the synthetic population data that is indexed by household ID.
`key – BTreeEntry *` – A BTreeEntry structure that contains the household ID as the first key.

Return Value A pointer to a static const `SyntheticHHData` Structure on success.
0 on error.

1.1.8 getSyntheticPopDemographicHeaders

Signature
```
int getSyntheticPopDemographicHeaders (FILE * const fp,
const char ** hh_header, const char ** person_header)
```

Description Reads a character array that holds the headers for both the household demographics and the person demographics. The headers are stored in the `hh_header` and `person_header` character arrays.

Argument `fp –x FILE *` pointer to the synthetic population. Must be open for reading.
Outputs `hh_header` – address of a character array that holds the headers for the household demographics.
`person_header` – address of a character array that holds the headers for the person demographics.

Return Value 1 on success.
0 on error.

1.1.9 CreateDoublePopulationIndex

Signature
```
void CreateDoublePopulationIndex(const char * filename,
int field_pos1, enum act_key_types field_type1,
int field_pos2, enum act_key_types field_type2,
const char *extension1, const char *extension2,
int headerLines)
```

Description Creates two indexes where the first primary index is field position 1 (**.extension1*), the first secondary index is field position 2, the second primary index is field position 2 *(*.extension2*), and the second secondary index is field position 1 from the datafile.

Argument
`filename` – the name of the data file to be indexed.
`field_pos1` – the position of the field to be used for the first index.
`field_type1` – the type of the first field (i.e., `kTypeint` for an integer field.)
`field_type2` – the position of the field to be used for the second index.
`field_type2` – the type of the first field (i.e., `kTypeFloat` for a float point field.)
`extension1` – the extension for the first index (i.e., *hh.idx*).
`extension2` – the extension for the second index (i.e., *trv.idx*).
`headerLines` – the number of header lines in a population file.

Return Value No return value.

1.1.10 CreatePopIndexFromFile

Signature
```
BTree *  CreatePopIndexFromFile(const char * data_file),
const char * index_file, int fieldPos1,
enum act_key_types keyType1, int fieldPos2,
enum act_key_types keyType2)
```

Description Creates a population index from a data file.

Argument
`data_file` – the file to be indexed.
`index_file` – the index file.
`fieldPos1` – the position of the field to be used for the first index.
`keyType1` – the type of the first field (i.e., `kTypeInt` for an integer field).
`fieldPos2` – the position of the field to be used for the second index.
`keyType2` – the position of the second field (i.e., `kTypeInt` for an integer field).
`extension2` – the extension for the second index (i.e., *trv.idx*).
`headerLines` – the number of header lines in a population file.

Return Value No return value.

1.2 Data Structures

1.2.1 SyntheticPersonData

This structure is used to hold synthetic person information.

```
typedef struct synPersonData_s
{
/** TRANSIMS Person ID. **/
INT32 fPersonID;

/** Array of person demographic information. **/
INT32 *fPersonDemographics;

} SyntheticPersonData;
```

1.2.2 SyntheticHHData

This structure is used to hold synthetic household information.

```
typedef struct synHHdata_s
{

/** The Census Tract ID of the household. **/
INT32 fTract;

/** The Block group ID of the household. **/
INT32 fBlockGroupID;

/** The TRANSIMS Household ID. **/
INT32 fHHId;

/** The number of persons in the household. **/
int fNumberPersons;

/** The number of vehicles owned by the household. **/
int fNumberVehicles;

/** The home location of the household – a TRANSIMS activity
 *  location ID.
 **/
INT32 fHomeLocation;

/** Number of data items in the household
 *  demographics/data array. **/
int fNumberHHDemographics;

/** Array of household demographic/data information.
 **/
REAL *fHHDemographics;
```

```
/** Number of demographics in the person demographics array. **/
int fNumberPersonDemographics;

/** Array of synthetic person records, one for each member of the
 *  household.
 *  The number of valid entries in this array is given by
 *  the fNumberPersons field.
 */
SyntheticPersonData *fPersons;

} SyntheticHHData;
```

2. ACTIVITIES

2.1 Interface Functions

The activity subsystem has C structures and utility functions that are used to read and write activity data from a TRANSIMS activity file. These functions assume that all of the activities for a household are grouped sequentially in the TRANSIMS activity file.

2.1.1 moreActivities

Signature `int moreActivities(FILE * const fp)`

Description Boolean function used to control iteration through the activity file.

Argument `fp – FILE *` for the activity file, which must be open for reading.

Return Value 1 if not at end of activity file.
0 if EOF has been reached.

2.1.2 getNextActivity

Signature `const ActivityData * getNextActivity(FILE * const)`

Description Reads an activity from the activity file. Parses and converts the string values from the file and stores them in a static `ActivityData` structure. Allocates storage for the `fOtherParticipantsList` and `fLocations` arrays based on data in the file.

Argument `fp – FILE *` to the activity, which must be open for reading.

Return Value The address of an unmodifiable `ActivityData` structure containing the activity data from the file. Returns `NULL` on error.

2.1.3 getNextHousehold

Signature
```
const ActivityData * getNextHousehold(FILE * const fp,
int* arraySize)
```

Description Reads the activities for a household from the activity file. Constructs an `ActivityData` structure for each activity in the household. Parses the activities and stores them in an array of `ActivityData` structures.

Argument `fp – FILE *` to the activity file, which must be open for reading.

Return Value An array of unmodifiable `ActivityData` structures that contains the activity data for the household. Returns `NULL` on error. The number of activities for the household is returned in the `arraySize` argument.

2.1.4 writeActivity

Signature
```
int writeActivity(FILE * const fp,
const ActivityData * data)
```

Description Writes the given `ActivityData` into a line of the given activity file.

Argument `fp` – `FILE *` to the activity file, which must be open for writing
`data` – address of an `ActivityData` structure containing the data to be written.

Return Value 1 on success.
0 on error.

2.1.5 writeHousehold

Signature
```
int writeHousehold(FILE * fp,
ActivityData * data, int arraySize)
```

Description Writes the activities for a household into the given file.

Argument `fp` – `FILE*` to the activity file, which must be open for writing.
`data` – address of an `ActivityData` array containing the household activity data to be written.
`arraySize` – the number of activities in the data array.

Return Value 1 on success.
0 on error.

2.1.6 moreTripTableEntries

Signature
```
int moreTripTableEntries(FILE * const)
```

Description Boolean function used to control iteration through the trip table file.

Argument `fp` – `FILE*` to the trip table file, which must be open for reading.

Return Value 1 if not at end of trip table file.
0 if EOF has been reached.

2.1.7 getTripTableDimensions

Signature `void` **`getTripTableDimensions`**`(FILE *fp const, int *x, int *y)`

Description Returns the x and y dimensions of the trip table.

Argument `fp` – `FILE*` to the trip table file, which must be open for reading.
`x` – the address of an integer that will contain the x dimension of the trip table.
`y` – the address of an integer that will contain the y dimension of the trip table.

Return Value The dimensions of the trip table are returned in the x and y arguments.

2.1.8 getNextTripTableEntry

Signature `const TTripTableEntry*` **`getNextTripTableEntry`**`(FILE * const)`

Description Reads the next trip table entry from the trip table file. Stores the information in a static `TTripTableEntry` structure and returns the address of this structure.

Argument `fp` – `FILE*` to the trip table file, which must be open for reading.

Return Value The address of a static `TTripTableEntry` structure that contains the data for the entry.
Returns `NULL` on error.

2.1.9 moreTimeTableEntries

Signature `int` **`moreTimeTableEntries`**`(FILE * const)`

Description Boolean function used to control iteration through the time table file.

Argument `fp` – `FILE*` to the time table file, which must be open for reading.

Return Value 1 if not at end of time table file.
0 if EOF has been reached.

2.1.10 getNextTimeTableEntry

Signature `const` **`getNextTimeTableEntry`**`(FILE * const)`

Description Reads the next time table entry from the time table file. Stores the

information in a static `TTimeTableEntry` structure and returns the address of this structure.

Argument `fp – FILE*` to the time table file, which must be open for reading.

Return Value The address of a static `TTimeTableEntry` structure that contains the data
for the entry.
Returns `NULL` on error.

2.1.11 CreateActivityIndex

Signature `void` **`CreateActivityIndex`**`(const char * actFileName)`

Description Creates a household and traveler index for a TRANSIMS activity file. The household index has the household id as the primary key and produces the file *<actFileName>.hh.idx*. The traveler index has the traveler id as the primary key and produces the file *<actFileName>trv.indx*.

Argument `actFileName` – the name of the TRANSIMS activity file.

2.1.12 moreSurveyActivities

Signature `int` **`moreSurveyActivities`**`(FILE *fp const)`

Description Boolean function used to control iteration through a survey activities file.

Argument `fp – FILE *` for the survey activity, which must be open for reading.

Return Value 1 if not at end of survey activity file
0 if EOF has been reached.

2.1.13 readSurveyActivityHeader

Signature `int` **`readSurveyActivityHeader`**`(FILE *fp const)`

Description Reads the header line in the survey activity file.

Argument `FILE *` for the survey activity, which must be open for reading.

Return Value 1 on success.
0 on error.

2.1.14 getSurveyActivity

Signature `const TSurveyActivityEntry* getSurveyActivity (FILE *fp const)`

Description Reads a survey activity from the survey activity file and stores the data in a static `TSurveyActivityEntry` structure.

Argument `fp – FILE *` to the survey activity file, which must be open for reading.

Return Value The address of an unmodifiable `TSurveyActivityEntry` structure containing the survey activity data from the file.
Returns `NULL` on error.

2.1.15 getSurveyWeightFromFile

Signature `const TSurveyWeightEntry* getSurveyWeightFromFile (FILE * const)`

Description Reads a survey weight entry from the file and stores the data in a static `TSurveyWeightEntry` structure.

Argument `fp – FILE *` to the survey weight file, which must be open for reading.

Return Value The address of an unmodifiable `TSurveyWeightEntry` structure containing the survey activity data from the file.
Returns `NULL` on error.

2.1.16 getSurveyWeightFromData

Signature `const TSurveyWeightEntry* getSurveyWeightFromData (char * const)`

Description Get a survey weight entry from the data pointer and store the data in a static `TSurveyWeightEntry` structure.

Argument `fp – char *` to the data character buffer.

Return Value The address of an unmodifiable `TSurveyWeightEntry` structure containing the survey activity data from the file.
Returns `NULL` on error.

2.1.17 moreTravelTimes

Signature `int` **`moreTravelTimes`**`(FILE * const)`

Description Boolean function used to control iteration through a travel times file.

Argument `fp – FILE 8*` for the travel times file, which must be open for reading.

Return Value 1 if not at end of survey activity file
0 if EOF has been reached.

2.1.18 getTravelTimeEntryFromFile

Signature `const TTravelTimeEntry*` **`getTravelTimeEntryFromFile`**
`(FILE * const)`

Description Reads a travel time entry from the file and stores the data in a static `TTravelTimeEntry` structure.

Argument `fp – FILE *` to the travel time file, which must be open for reading.

Return Value The address of an unmodifiable `TTravelTimeEntry` structure containing the travel time data from the file.
Returns `NULL` on error.

2.1.19 getTravelTimeEntryFromData

Signature `const TTravelTimeEntry*` **`getTravelTimeEntryFromData`**
`(char * const)`

Description Get a travel time entry from the data pointer and store the data in a static `TTravelTimeEntry` structure.

Argument `fp – char *` to the data character buffer.

Return Value The address of an unmodifiable `TTravelTimeEntry` structure containing
the travel time data.
Returns `NULL` on error.

2.1.20 writeTravelTimeEntry

Signature `int` **`writeTravelTimeEntry`**`(const TTravelTimeEntry * const)`

Description Write a travel time entry to a file.

Argument `entry –` to a `TTravelTimeEntry` structure containing the data to be written to the file.
`fp – FILE *` to the file where the entry will be written; must be open for writing.

Return Value 1 on success.
0 on error.

2.1.21 moreTreeEntries

Signature `int` **`moreTreeEntries`**`(FILE * const)`

Description Boolean function used to control iteration through a decision tree file.

Argument `fp – FILE *` for the decision tree file, which must be open for reading.

Return Value 1 if not at the end of decision tree file.
0 if EOF has been reached.

2.1.22 getTreeEntryFromFile

Signature `const TTreeEntry *` **`getTreeEntryFromFile`**`(FILE * const)`

Description Reads a decision tree entry from the file and stores the data in a static `TTreeEntry` structure.

Argument `fp – FILE *` to the decision tree file, which must be open for reading.

Return Value The address of an unmodifiable `TTreeEntry` structure containing the decision tree data from the file.
Returns `NULL` on error.

2.1.23 getTreeEntryFromData

Signature `const TTreeEntry` **`getTreeEntryFromData`**`(char * const)`

Description Get a decision tree entry from the data pointer and store the data in a static `TTreeEntry` structure.

Argument `fp – char *` to the data character buffer.

Return Value The address of an unmodifiable `TTreeEntry` structure containing the

decision tree data.
Returns `NULL` on error.

2.1.24 moreZoneEntries

Signature `int **moreZoneEntries**(FILE * const)`

Description Boolean function used to control iteration through a zone information file.

Argument `fp – FILE *` for the zone information file, which must be open for reading.

Return Value 1 if not at end of zone information file.
0 if EOF has been reached.

2.1.25 getZoneHeaderFromFile

Signature `const TZoneHeader * **getZoneHeaderFromFile**(FILE * const)`

Description Reads the zone header from the file and stores the data in a static `TZoneHeader` structure.

Argument `fp – FILE *` to the zone information file, which must be open for reading.

Return Value The address of an unmodifiable `TZoneHeader` structure containing the header data from the file.
Returns `NULL` on error.

2.1.26 getZoneEntryFromFile

Signature
```
const TZoneEntry * getZoneEntryFromFile
FILE * const, int)
```

Description Reads a zone entry from the file and stores the data in a static `TZoneEntry` structure.

Argument `fp – FILE *` to the zone information file, which must be open for reading.

Return Value The address of an unmodifiable `TZoneEntry` structure containing the zone data from the file.
Returns `NULL` on error.

2.1.27 getZoneEntryFromData

Signature `const TZoneEntry * getZoneEntryFromData (char * const, int)`

Description Get a zone entry from the data pointer and store the data in a static `TZoneEntry` structure.

Argument `fp – char *` to the data character buffer.

Return Value The address of an unmodifiable `TZoneEntry` structure containing the zone data.
Returns `NULL` on error.

2.1.28 moreModeWeightEntries

Signature `int moreModeWeightEntries(FILE * const)`

Description Boolean function used to control iteration through a mode weight file.

Argument `fp – FILE *` for the mode weight file, which must be open for reading.

Return Value 1 if not at end of mode weight file.
0 if EOF has been reached.

2.1.29 getModeWeightEntryFromData

Signature `const TModeWeightEntry * getModeWeightEntryFromData (char * const)`

Description Get a mode coefficient entry from the data pointer and store the data in a static `TModeWeightEntry` structure.

Argument `fp – char *` to the data character buffer.

Return Value The address of an unmodifiable `TModeWeightEntry` structure containing the zone data.
Returns `NULL` on error.

2.1.30 getModeWeightEntryFromFile

Signature `const TModeWeightEntry * getModeWeightEntryFromFile (FILE * const)`

Description Reads a mode weight entry from the file and stores the data in a static

`TModeWeightEntry` structure.

Argument `fp – FILE *` to the mode weight file, which must be open for reading.

Return Value The address of an unmodifiable `TModeWeightEntry` structure containing the mode weight data from the file.
Returns `NULL` on error.

2.1.31 moreModeEntries

Signature `int` **`moreModeEntries`**`(FILE * const)`

Description Boolean function used to control iteration through a TRANSIMS mode file.

Argument `fp – FILE *` for the mode file, which must be open for reading.

Return Value 1 if not at end of mode.
0 if EOF has been reached.

2.1.32 getModeEntryFromFile

Signature `const TModeEntry *` **`getModeEntryFromFile`**`(FILE * const)`

Description Reads a mode entry from the file and stores the data in a static `TModeEntry` structure.

Argument `fp – FILE *` to the mode file, which must be open for reading.

Return Value The address of an unmodifiable `TModeEntry` structure containing the mode data from the file.
Returns `NULL` on error.

2.1.33 CreateFeedbackIndex

Signature `void` **`CreateFeedbackIndex`**`(const char * FileName`

Description Creates an index to the feedback command file with household ID as the primary key and traveler ID as the secondary key.

Argument `FileName` – name of the feedback command file to be indexed.

2.1.34 CreateTravelTimesIndex

Signature `void **CreateTravelTimesIndex**(const char * FileName)`

Description Creates an index to the travel times file with zone 1 as the primary index and zone 2 as the secondary index.

Argument `FileName` – name of the travel times file to be indexed.

2.2 Data Structures

2.2.1 ActivityTimeSpec

This structure is used for activity time specifications.

```
typedef struct act_time_spec_s
{
/** The lower bound of the time interval. **/
REAL fLowerBound;

/** The upper bound of the time interval. **/
REAL fUpperBound;

/** The A parameter for the beta distribution. **/
REAL fAParameter;

/** The B parameter for the beta distribution. **/
REAL fBParameter;

} ActivityTimeSpec;
```

Each activity has a start time, end time, and duration range. The preferred time for each of these is given in terms of the two parameters of a beta distribution, $f(t)=C(t-L)^{a-1}(U-t)^{b-1}$, where C is a constant, L is the lower bound of the time, U is the upper bound and a and b are the parameters that specify the distribution. The mean of the distribution is $a/(a+b)$; $a=1$ and $b=1$ gives a uniform distribution between L and U. Larger values for a and b result in a more peaked distribution. If the a and/or b parameter is equal to -1.0, an average of the lower and upper bound will be used.

The reference time is taken as 0.00 (midnight of the first day). All times are decimal numbers that denote the number of hours from 0.00. Note that each time should be given to a minimum of two decimal places to capture minutes and four decimal places if seconds are necessary.

2.2.2 ActivityData

This structure is used to store the data for a single activity as defined by one line in the activity file.

```
typedef struct actdata_s
{
/** The household ID. **/
INT32 fHouseholdId;

/** The person ID. **/
INT32 fPersonId;

/** The activity ID – must be unique within the household. **/
INT32 fActivityId;

/** Activity type.  An integer value representing
 *  the activity type such as home, work, school, shopping,
 *  other, wait at transit stop, ... .
**/
INT32 fType;

/** Priority ranking of the activity in the range 0 - 9,
 *  where 0 is the lowest priority and 9 means the activity
 *  must be done.
**/
INT32 fPriority;

/** Integer value defining transportation mode used to arrive
 *  at the activity.
**/
INT32 fModePreference;

/** The ID of the vehicle to be used when the mode preference is
 *  private auto, either as a driver or passenger.  Set to –1 for
 *  all other mode preferences.
**/
INT32 fVehicleId;

/** The number of locations where the activity can take place.
 *  This field is used to provide information about the
 *  fActivityGroupIndex and fPossibleLocationsList fields.
 *  A value of 1 or greater indicates that the
 *  fPossibleLocationsList contains a list of locations for the
 *  activity. A value of -1 indicates that the
 *  fActivityGroupIndex field contains an index number into a
 *  group of activities.
**/
INT32 fPossibleLocations;

/** The number of other people that will participate in the
```

```
 *  activity and use the same transportation. Value is 0 if the
 *  person is traveling alone to the activity. If the value is >
 *  0, a list of the IDs of the other participants is entered in
 *  the fOtherParticipantsList array.
**/
INT32 fOtherParticipants;

/** Number of the activity for this individual. Every activity
 *  for an individual has a number. Groups of activities that
 *  must be done together have the same number.
**/
INT32 fActivityGroupNumber;

/** An array of personIds for other participants in the activity
 *  that will use the same transportation. There are no valid
 *  entries in this array if the value of fOtherParticipants
 *  is 0.
**/
INT32 *fOtherParticipantsList;

/** Index into a group of activities (integer).
 *  Used only when value of fPossibleLocations is -1.
**/
INT32 fActivityGroupIndex;

/** An array of possible locations (integer IDs) where
 *  the activity will occur. Used when value of
 *  fPossibleLocations is 1 or greater.
**/
INT32 *fLocations;

/** Preferred start time for the activity.  The ActivityTimeSpec
 *  structure contains the specification parameters for a beta
 *  distribution of the preferred time.
**/
ActivityTimeSpec fStart;

/** Preferred end time for the activity.  The ActivityTimeSpec
 *  structure contains the specification parameters for a beta
 *  distribution of the preferred time.
**/
ActivityTimeSpec fEnd;

/** Preferred duration for the activity.  The ActivityTimeSpec
 *  structure contains the specification parameters for a beta
 *  distribution of the preferred time.
**/
ActivityTimeSpec fDuration;

} ActivityData;
```

2.2.3 TTripTableEntry

This structure is used to store a two-dimensional table containing the number of trips between zones.

```
typedef struct triptableentry_s
{
/** The number of the X zone. **/
int fZoneX;

/** The number of the Y zone. **/
int fZoneY;

/** The number of trips between fZoneX and fZoneY **/
int fNumberTrips;

} TTripTableEntry;
```

2.2.4 TTimeTableEntry

This structure is used to store entries from a trip time probability table that contains a range of times over a 24-hour period. Each range has an associated trip probability.

```
typedef struct timetableentry_s
{
/** The lower bound of the time range. **/
float fRangeL;

/** The upper bound of the time range. **/
float fRangeU;

/** The probability associated with the time range. **/
float fProb;


} TTimeTableEntry;
```

2.2.5 TSurveyActivityEntry

This structure stores the data for a survey activity as defined by one line in the survey activity file.

```
/** A survey activity entry **/

typedef struct TSurveyActivityEntry_s
{
/** Survey household number. **/
INT32 fHHNumber;

/** Person number (unique within the household) **/
INT32 fPersonNumber;
```

```
/** Activity number for each person, 0 - n.
 *  0 = initial at home activity.
 */
INT32 fActivityNumber;

/** Activity type, 0 = at home, 1 = work, 22 = serve_passengers.
 * Others may vary.
 */
INT32 fType;

/** 1 if activity is at home location, 0 if out of home. **/
 INT32 fAtHome;

/** Value = 1 if person was already at the location,
 *  value = 2 if not.
 */
INT32 fWereYouThere;

/** Mode for arriving at activity.  -1 if mode from
 *  survey was missing.
 */
INT32 fMode;

/** Value = 1 if person was driver, 2 if person was a passenger,
 * -1 otherwise.
 */
INT32 fDriver;

/** Activity start time in minutes after midnight (0 - 2400). **/
INT32 fStartTime;

/** Activity end time in minutes after midnight (0 - 2400). **/
INT32 fEndTime;

/** Number of persons in vehicle **/
INT32 fNumberInVehicle;

/** X coordinate of survey activity **/
REAL fXCoord;

/** Y coordinate of survey activity **/
REAL fYCoord;

} TSurveyActivityEntry;
```

2.2.6 TTravelTimeEntry

This structure stores data for one zone-to-zone travel time entry as contained in one line of the travel times file.

```
/** An activity generator travel time entry.
```

```
 *  Travel time values are from zone to zone
 *  by mode and time of day.
 */
typedef struct TTravelTimeEntry_s
{
/** Zone numbers  **/
INT32 fZone1;
INT32 fZone2;

/** Mode **/
INT32 fMode;

/** Start Time **/
REAL fStartTime;

/**End Time **/
REAL fEndTime;

/** Travel time **/
REAL fValue;

/** Time entry was updated. **/
INT32 fLastUpdate;

} TTravelTimeEntry;
```

2.2.7 TFeedbackEntry

This structure is used to store the data from an activity generator feedback command.

```
/** An activity generator feedback file entry.
 *  An entry contains the household id,
 *  the activity id, the command, and
 *  an optional arguments to the command.
 *  If the command is change time of the activity,
 *  the new start, end, start alpha and beta,
 *  and end alpha and beta are specified as
 *  arguments.
 */
typedef struct TFeedbackEntry_s
{
/** Household ID **/
INT32 fHouseholdId;

/** Activity id **/
INT32 fActivityId;

/** The feedback command **/
char fCommand[MAX_FEEDBACK_COMMAND_LENGTH];

/** The number of valid arguments in the fArguments array. **/
int fValidArgs;
```

```
/** The optional arguments to the command. **/
REAL fArguments[MAX_NUMBER_FEEDBACK_ARGUMENTS];

} TFeedbackEntry;
```

2.2.8 TTreeEntry

This structure is used to store the data for a node in the Activity Generator regression tree.

```
/** An entry defining a node in the Activity Generator decision
 *  tree. Each node contains a demographic, a split value for the
 *  demographic, and a node number that specified it's
 *  relationship in the tree.
 */
typedef struct TTreeEntry_s
{
/** The demographic number. **/
INT32 fDemographic;

/** The split value for the demographic. **/
REAL fSplitValue;

/** The node number. **/
INT32 fNodeNumber;

} TTreeEntry;
```

2.2.9 TZoneHeader

This structure is used to store the header information from the Activity Generator zone information file.

```
/** Stores the column headings from the header line
 *  in the zone information data.
 *  The number of column headings is variable.
 */
typedef struct TZoneHeader_s
{

/** The number of columns in the line. **/
int fNumberHeaders;

/** Array of character strings containing the column
 *  headings for the attractor values in the zone.
 *  Each column header can have up to MAX_HEADER_LENGTH
 *  characters.  This is a dynamically allocated
 *  2-dimensional array  fAttractorHeaders[][]
 */
char **fAttractorHeaders;
```

```
} TZoneHeader;
```

2.2.10 TZoneEntry

This structure is used to store data for a zone entry from the Activity Generator zone information file.

```
typedef struct TZoneEntry_s
{
/** Zone number. **/
INT32 fNumber;

/** Easting geocoordinate for the zone. **/
 REAL fEasting;

/** Northing geocoordinate for the zone. **/
REAL fNorthing;

/** Number of attractors by activity type for the zone.
 *  The work attractor is required so the number must
 *  be 1 or greater.  These types MUST correspond to
 *  the activity type definitions in the Activity Generator.
 */
INT32 fNumberAttractors;

/** Array of floating point values for the attractors by
 *  activity type in the zone.  The first value in the
 *  array is for the work attractor which is required.
 */
REAL *fAttractors;

} TZoneEntry;
```

2.2.11 TModeWeightEntry

This structure is used to store a mode weight entry from the Activity Generator mode coefficient file.

```
/** A mode coefficient entry for the NISS activity generator.
 *  Each mode can be assigned a relative weight.
 */
typedef struct TModeWeightEntry_s
{

/** The coefficient for the mode. **/
REAL fWeight;

/** The activity type **/
INT32 fActivityType;
```

```
/** The mode **/
INT32 fMode;

} TModeWeightEntry;
```

2.2.12 TModeEntry

This structure is used to store an entry from a TRANSIMS mode map file.

```
/** A mode string and number entry.
 *  Each entry has a mode string, e.g. "wcw"
 *  and an integer value associated with the mode.
 */
typedef struct TModeEntry_s
{
/** The mode string **/
char fModeString[MAX_MODE_STRING_LENGTH];

/** The number associated with the mode string. **/
int fMode;

} TModeEntry;
```

3. VEHICLE

3.1 Interface Functions

The vehicle subsystem has C structures and utility functions that are used to read and write data from a TRANSIMS vehicle file.

The function **`getNextVehicle()`** reads vehicle data from a vehicle file in ASCII format. The function stores the information in an unmodifiable data structure (`VehicleData`), and returns a pointer to the structure. Since the `VehicleData` structure cannot be modified by the calling program, the data should be copied if it needs to be changed.

The function **`writeVehicle()`** takes a `VehicleData` structure as an argument containing the information to be written. The **`getNextVehicle()`** function combined with the **`moreVehicles()`** function provides a mechanism for iterating through the vehicle file reading the vehicle data.

3.1.1 moreVehicles

Signature `int moreVehicles(FILE * const fp)`

Description Boolean function used to control iteration through the vehicle file.

Argument `fp` – file pointer for the vehicle file, which must be open for reading.

Return Value 1 if not at end of vehicle file.
0 if EOF has been reached.

3.1.2 getNextVehicle

Signature `const VehicleData * getNextVehicle(FILE * const fp)`

Description Reads a line of vehicle data from the vehicle file. Parses and converts the string values from the file and stores them in the static `VehicleData` structure `fVehicle`.

Argument `fp` – file pointer for the vehicle file, which must be open for reading.

Return Value The address of a static `VehicleData` structure containing the vehicle data read from the file.
Returns `NULL` on error.

3.1.3 writeVehicle

Signature
```
int writeVehicle(FILE * const fp,
const VehicleData * data)
```

Description Writes the given `VehicleData` into a line of the given vehicle file.

Argument `fp` – file pointer for the vehicle file, which must be open for reading.
`data` – address of a `VehicleData` structure containing the data to be written.

Return Value 1 on success.
0 on error.

3.1.4 VehDataReadHeader

Signature
```
int VehDataReadHeader(FILE * fp,
TVehDataHeader * header)
```

Description Reads the header line from the vehicle file.

Argument `fp` – file pointer for the vehicle file, which must be opened for reading.
`header` – `TVehDataHeader *` to a header structure.

Return Value 1 if header read successfully.
0 if error occurs.

3.1.5 VehDataWriteHeader

Signature
```
int VehDataWriteHeader(FILE * fp,
TVehDataHeader * header)
```

Description Writes a header line to the vehicle file.

Argument `fp` – file pointer for the vehicle file, which must be opened for writing.
`header` – `TVehDataHeader *` to a header structure.

Return Value 1 if header written successfully.
0 if error occurs.

3.1.6 VehDataWriteDefaultHeader

Signature
```
int VehDataWriteDefaultHeader(FILE * fp)
```

Description Writes a default header line to the vehicle file.

Argument `fp` – file pointer for the vehicle file, which must be opened for writing.

Return Value 1 if header written successfully.
0 if error occurs.

3.1.7 VehDataSkipHeader

Signature `int VehDataSkipHeader(FILE * fp)`

Description Skip a header from a vehicle file.

Argument `fp` – file pointer for the vehicle file, which must be opened for reading.

Return Value 1 if header skipped successfully.
0 if error occurs.

3.2 Data Structures

3.2.1 TVehDataHeader

This structure is used to store a vehicle file header.

```
typedef struct
{
/** The field names. **/
INT8 fFields[512]

} TVehDataHeader;
```

3.2.2 VehicleData

This structure is used to store the data for a single vehicle as defined by one line in the vehicle file.

```
typedef struct vehdata_s
{
/** The household ID. **/
INT32 fHouseholdId;

/** The vehicle ID. **/
INT32 fVehicleId;

/** The ID starting location of the vehicle.  -1 is used if
 *  the starting location is unknown or to indicate that the
 *  route planner should choose the starting location.
**/
INT32 fStartingLocation;
```

```
/** The TRANSIMS Network vehicle type.
 *  Must be one of the following values:
 *    1 = Auto
 *    2 = Truck
 *    4 = Taxi
 *    5 = Bus
 *    6 = Trolley
 *    7 = StreetCar
 *    8 = LightRail
 *    9 = RapidRail
 *   10 = RegionalRail
 *   -1 = Unknown
**/
INT32 fVehicleType;

/** The user-defined emissions vehicle subtype. **/
INT32 fEmissionsSubtype;

/** The number of values in the fIdentifiers array. **/
INT32 fNumberIdentifiers;

/** Optional array of user defined integer values.
 *  The number of entries in the array is variable
 *  but must be the same for every line of the file.
 *  If no user-defined values are present in the file,
 *  fIdentifiers will be NULL.
**/
INT32 *fIdentifiers;

} VehicleData;
```

3.2.3 TVehDataHeader

This structure is used to store the data for a vehicle location file.

```
typedef struct
{
/* The field names. */
INT8 fFields[512];

} TVehDataHeader;

/** Read a header from a vehicle location file.
 *  Return nonzero if the header was
 *  successfully read, or zero if not. **/

extern int VehDataReadHeader(FILE* file, TVehDataHeader* header);

/** Write a header to a vehicle location file.
 *  Return nonzero if the header was
```

```
 *  successfully written, or zero if not. **/
extern int VehDataWriteHeader(FILE* file,
const TVehDataHeader* header);

/** Write a default header to a vehicle location file.
 *  Return nonzero if the header was successfully written,
 *  or zero if not. **/
extern int VehDataWriteDefaultHeader(FILE* file);

/** Skip a header from a vehicle location file.
 *  Return nonzero if the header was successfully skipped,
 *  or zero if not. **/
extern int VehDataSkipHeader(FILE* file);
}
```

3.3 Files

Table 1: Vehicle File library files.

Type	File Name	Description
Binary Files	*libTIO.a*	TRANSIMS Interfaces library
Source Files	*vehio.c*	Source file for vehicle file functions
	vehio.h	Header file for vehicle file functions

4. VEHICLE PROTOTYPES

This section describes the C structures and utility functions that are used to read and write TRANSIMS vehicle prototype files. Vehicle prototype files are used to describe parameters for vehicle types and subtypes such as length and capacity of vehicle, and maximum speed and acceleration.

4.1 Interface Functions

4.1.1 VehReadHeader

Signature `int` **`VehReadHeader`**`(FILE * file, TVehHeader * header)`

Description Read a header from a vehicle prototype file.

Argument `file` – pointer to `FILE` stream object.
`header` – pointer to a vehicle prototype header structure.

Return Value Returns nonzero if the header was successfully read, or zero if not.

4.1.2 VehWriteHeader

Signature
```
int VehWriteHeader(FILE * file,
const TVehHeader * header)
```

Description Write a header from a vehicle prototype file.

Argument `file` – pointer to `FILE` stream object.
`header` – pointer to a vehicle prototype header structure.

Return Value Returns nonzero if the header was successfully written, or zero if not.

4.1.3 VehWriteDefaultHeader

Signature `int` **`VehWriteDefaultHeader`**`(FILE * file)`

Description Write a default header from a vehicle prototype file.

Argument `file` – pointer to `FILE` stream object.

Return Value Returns nonzero if the header was successfully written, or zero if not.

4.1.4 VehSkipHeader

Signature `int VehSkipHeader(FILE * file)`

Description Skip a header from a vehicle prototype file.

Argument `file` – pointer to FILE stream object.

Return Value Returns nonzero if the header was successfully shipped, or zero if not.

4.1.5 VehReadPrototype

Signature
```
int VehReadPrototype(FILE * file,
TVehPrototypeData * record)
```

Description Read a record from a vehicle prototype file.

Argument `file` – pointer to `FILE` stream object.
`record` – pointer to a vehicle prototype record structure.

Return Value Returns nonzero if the record was successfully read, or zero if not.

4.1.6 VehWritePrototype

Signature
```
int VehWritePrototype(FILE * file,
const TVehPrototypeData *record)
```

Description Write a record to a vehicle prototype file.

Argument `file` – pointer to `FILE` stream object.
`record` – pointer to a vehicle prototype record structure.

Return Value Returns nonzero if the record was successfully written, or zero if not.

4.2 Data Structures

4.2.1 TVehHeader

This structure is used to store the vehicle prototype as defined by one line in the vehicle file.

```
typedef struct
{
/** The field names. **/
INT8 fFields[512]
```

```
} TVehHeader;
```

4.2.2 TVehPrototypeData

This structure is used for vehicle prototype file records.

```
typedef struct
{
/** The vehicle type. **/
{
/** The vehicle type. **/
INT32 fVehicleType;

/** The vehicle subtype, used for emissions. **/
INT32 fEmissionsSubtype;

/** The maximum vehicle speed (meters/second). **/
REAL fMaximumVelocity;

/** The maximum vehicle acceleration (meters/second/seconds). **/
REAL fMaximumAcceleration;

/** The vehicle length (meters). **/
REAL fLength;

/** The vehicle capacity (driver + number of possible
 *  passengers).
 **/
INT32 fCapacity;

}TVehPrototypeData;
```

4.3 Files

Table 2: Vehicle Prototype File library files.

Type	File Name	Description
Binary Files	*libTIO.a*	TRANSIMS Interfaces library
Source Files	*vehprotoio.c*	Source file for vehicle prototype file functions
	vehprotoio.h	Header file for vehicle prototype file functions

5. PLAN

The Route Planner and Traffic Microsimulator have C structures and utility functions that are used to read and write TRANSIMS plan files.

5.1 Interface Functions

The function **`getNextLegRecord()`** reads a single leg of a traveler's plan from a plan file. The function stores the information in a static data structure (LegData) and returns a pointer to a structure data. The `LegData` structure cannot be modified by the calling program. The data should be copied if it needs to be changed. The function **`writeLeg()`** is used to create a TRANSIMS plan file.

5.1.1 moreLegs

Signature `int` **`moreLegs`**`(FILE * const fp)`

Description Boolean function used to control iteration through the plan file.

Argument `fp` – `FILE *` for the plan file, which must be open for reading.

Return Value 1 if not at end of plan file.
0 if EOF has been reached.

5.1.2 getNextLeg

Signature `const LegData *` **`getNextLeg`**`(FILE * const fp)`

Description Reads a single leg of a traveler's plan from the plan file. Parses and converts the non-mode dependent values from the file and stores them in the static `LegData` structure.

Argument `fp – FILE *` for the plan file, which must be open for reading.

Return Value The address of the static `LegData` structure containing the data read from the file.
Returns `NULL` on error.

5.1.3 writeLeg

Signature `int` **`writeLeg`**`(FILE * const fp, const LegData * leg)`

Description Writes the given `LegData` into the given plan file.

Argument `fp – FILE *` to the plan file, which must be open for reading.

Return Value 1 on success.
0 on error.

5.1.4 readLegRecordFromString

Signature `int readLegRecordFromString(char * buf)`

Description Reads and parses the data for a `LegData` record from a data string. The data string need not be null terminated.

Argument `buf` – the data character string.

Return Value 1 always.

5.1.5 readLegRecord

Signature `int readLegRecord(FILE * const fp)`

Description Reads a single leg of a traveler's plan from the plan file and stores it in a static character buffer. Note that it does not parse the record, nor does it update the contents of the static `LegData` structure. Selective parsing of records provides a fast means of retrieving only a few pieces of data from each leg of a plan file.

Argument `fp – FILE *` for the plan file, which must be open for reading.

Return Value 1 on success.
0 on error.

5.1.6 writeLegRecord

Signature `int writeLegRecord(FILE * const fp)`

Description Writes the current string stored in the static character buffer into the given plan file.

Argument `fp – FILE *` to the plan file, which must be open for writing.

Return Value Number of characters written.

5.1.7 parseBufferedLegRecord

Signature `int` **`parseBufferedLegRecord`**`(void)`

Description Parses all of the non-mode-dependent fields of the traveler's leg data stored in the static character buffer. Fills in the fields of the static `LegData` structure.

Return Value 1 always.
0 on failure.

5.1.8 getCurrentLeg

Signature `const LegData *` **`getCurrentLeg`**`(void)`

Description Retrieve the information currently stored in the static `LegData` structure.

Return Value The address of the static LegData structure maintained by **`parseBufferedLegRecord`**, **`readLegRecordFromString`**, and **`readLegRecord`**.

5.1.9 getLegTravelerId

Signature `int` **`getLegTravererId`**`(void)`

Description Parses the traveler ID field from the static character buffer and stores the result in the traveler ID field of the static `LegData` structure.

Return Value Traveler ID from the current leg.

5.1.10 getLegDepartTime

Signature `int` **`getLegDepartTime`**`(void)`

Description Parses the estimated departure time field from the static character buffer and stores the result in the estimated departure time field of the static `LegData` structure.

Return Value Estimated departure time from the current leg.

5.1.11 getLegStartAccessoryId

Signature `int` **`getLegStartAccessoryId`**`(void)`

Description Parses the starting accessory ID field from the static character buffer and stores the result in the starting accessory ID field of the static `LegData` structure.

Return Value Starting accessory ID from the current leg.

5.1.12 getLegStartAccessoryType

Signature `int getLegStartAccessoryType(void)`

Description Parses the starting accessory type field from the static character buffer and stores the result in the starting accessory type field of the static `LegData` structure.

Return Value Starting accessory type from the current leg.

5.1.13 getLegTripId

Signature `int getLegTripId()`

Description Parses the trip ID field from the static character buffer and stores the result in the trip ID field of the static `LegData` structure.

Return Value Trip ID from the current leg.

5.1.14 getLegLegId

Signature `int getLegLegId(void)`

Description Parses the leg ID from the static character buffer and stores the result in the leg ID field of the static `LegData` structure.

Return Value Leg ID from the current leg

5.1.15 getLegMode

Signature `int getLegMode(void)`

Description Parses the mode field from the static character buffer and stores the result in the mode field of the static `LegData` structure.

Return Value Mode from the current leg.

5.1.16 DefragmentPlanFiles

Signature `void` **`DefragmentPlanFile`**`(char * filename, BTree * index)`

Description Create one new plan file from an index that specifies multiple plan files.

Argument `filename` – pointer to a character string containing the name of the new plan file.
`index` – pointer to the existing index.

Return Value No value returned.

5.1.17 ReverseIndex

Signature `void` **`ReverseIndex`**`(BTree * new, BTree * old)`

Description Create a new index from an existing index by interchanging the primary and secondary keys.

Argument `new` – pointer to the new index, should have been prepared with `BTree_Create` prior to running **`ReverseIndex`**.
`old` – pointer to the existing index.

Return Value No value returned.

5.2 Data Structures

5.2.1 LegData

This structure is used to hold one leg of a traveler's plan.

```
typedef struct plandata_s
{
/** the fixed part of the data structure **/

/** The TRANSIMS traveler ID **/
UINT32 fTravId;

/** A user defined field **/
INT32 fUser;

/** The trip sequence number **/
INT32 fTrip;

/** The leg sequence number **/
INT32 fLeg;
```

```
/** The (estimated) departure time in seconds past midnight **/
INT32 fActivationTime;

/** The ID of the accessory at which this leg starts **/
INT32 fStartAcc;

/** The type of the accessory at which this leg starts **/
INT32 fStartAccType;

/** The ID of the accessory at which this leg ends **/
INT32 fEndAcc;

/** The type of the accessory at which this leg ends **/
INT32 fEndAccType;

/** The (estimated) duration of this leg in seconds **/
INT32 fDuration;

/** The (estimated) ending time of this leg in seconds past
 *  midnight
**/
INT32 fStopTime;

/** A flag telling the microsimulator whether to use the
 *  maximum or minimum of fStopTime and (fDuration + simulation
 *  time at arrival) for determining the ending time of a leg
 *  using a non-simulated mode
**/
INT32 fMaxTime;

/** The dollar cost of this leg **/
INT32 fCost;

/** The cost of this leg under a user-defined generalized cost
 *  function **/
INT32 fGCF;

/** Is the traveler driving a vehicle on this leg? **/
INT32 fDriverFlag;

/** The travel mode for this leg **/
INT32 fMode;

/* The variable (mode dependent) part of the data structure */
char * fData;

} LegData;
```

6. ITERATION DATABASE

6.1 Interface Functions

In any function that takes a string representing a record as an argument, an empty field is represented by two consecutive commas (i.e., ",,"). In any function that takes an array of strings representing a record as an argument, a blank field can be represented by either an empty string or a `NULL` pointer to a string. The last pointer in the array should be `NULL`.

The *String* functions return a null-terminated string that is a copy of the record/field requested. The contents of the string are modifiable, and the string needs to be freed after use.

If a particular field is empty, it is assumed that the value for that field has not changed since the last iteration.

The *Data* functions return a pointer into the mmapped file in which the record/field resides. Changing data through this pointer will change the actual iteration file where the data resides. This pointer should not be freed.

6.1.1 ITDB_Create

Signature `ITDB* ITDB_Create(char* base_filename, char* fields)`

Description Creates a new iteration database.

Argument `base_filename` – filename to create files with, *filename.idx* for the index and *filename.#.it* for each iteration, where # is the iteration number.
`fields` – field names as a comma-separated string.

Return Value A pointer to a new iteration database on iteration 0.

6.1.2 ITDB_CreateV

Signature
```
ITDB * ITDB_CreateV(char* base_filename,
char* fields[], int key)
```

Description Creates a new iteration database with the given filename.

Argument `base_filename` – filename to create files with, *filename.idx* for the index and *filename.#.it* for each iteration, where # is the iteration number.
`fields` – field names as an array of strings.
`key` – value of the key for which to return records.

Return Value A pointer to a new iteration database on iteration 0.

6.1.3 ITDB_Open

Signature `ITDB* ITDB_Open(char* base_filename)`

Description Opens an existing ITDB.

Argument `base_filename` – the filename of the ITDB.

Return Value A pointer to an existing iteration database on the same iteration it had when closed.

6.1.4 ITDB_Close

Signature `void ITDB_Close(ITDB* db)`

Description Closes an ITDB and free all resources. Upon return, `db` is no longer a valid pointer.

Argument `db` – the database to close.

Return Value None.

6.1.5 ITDB_CurrentIteration

Signature `int ITDB_CurrentIteration(ITDB* db)`

Description Returns the current iteration number.

Argument `db` – the itdb on which to operate.

Return Value Current iteration number.

6.1.6 ITDB_NewIteration

Signature `int ITDB_NewIteration(ITDB* db, char* comment)`

Description Starts on a new iteration.

Argument `db` – the itdb on which to operate.
`comment` – comment to be stored as the first line of the new iteration file.

Return Value New iteration number.

6.1.7 ITDB_Add

Signature `void` **`ITDB_Add`**`(ITDB* db, int key, char* data)`

Description Adds data to `key` for the current iteration. If data exists for the key given, the new data is added to the index following the old data.

Argument `db` – the itdb on which to operate.
`key` – value of primary key.
`data` – a comma-separated string of field values.

Return Value None.

6.1.8 ITDB_AddV

Signature `void` **`ITDB_AddV`**`(ITDB*, int key, char* data[])`

Description Adds data to `key` for the current iteration. If data already exists for the key given, the new data is added to the index following the old data.

Argument `db` – the itdb on which to operate.
`key` – value of primary key.
`data` – an array of field values.

Return Value None.

6.1.9 ITDB_GetCurrentString

Signature `char*` **`ITDB_GetCurrentString`**`(ITDB* db, int key)`

Description Get data for `key` from the current iteration.

Argument `db` – the itdb in which to operate.
`key` – value of key for which to retrieve information.

Return Value Null-terminated copy of the data. The caller is responsible for deleting this string.

6.1.10 GetCurrentData

Signature `char*` **`ITDB_GetCurrentData`**`(ITDB* db, int key)`

Description Get data for `key` from the current iteration.

Argument `db` – the itdb on which to operate.
`key` – value of key for which to retrieve data.

Return Value A pointer into the mmapped field. Changes to the string will change the actual file. This pointer should not be freed.

6.1.11 ITDB_GetString

Signature `char* ITDB_GetString(ITDB* db, int it, int key)`

Description Get data for `key` from the given iteration.

Argument `db` – the itdb on which to operate.
`it` – iteration from which to retrieve data.
`key` – value of key for which to retrieve information.

Return Value Null-terminated copy of the data. The caller is responsible for deleting this string.

6.1.12 ITDB_GetData

Signature `char* ITDB_GetData(ITDB* db, int it, int key)`

Description Get data for `key` from the given iteration.

Argument `db` – the itdb on which to operate.
`it` – iteration from which to retrieve data.
`key` – value of key for which to retrieve information.

Return Value A pointer into the mmapped file. Changes to the string will change the actual file. This pointer should not be freed.

6.1.13 ITDB_GetTotalString

Signature `char* ITDB_GetTotalString(ITDB* db, int key)`

Description Returns the latest data over all iterations for `key`. Searches back through the iterations for the last non-blank entry for each field.

Argument `db` – the itdb on which to operate.
`key` – value of key for which to retrieve information.

Return Value Null-terminated copy of the data. The caller is responsible for deleting this string.

6.1.14 ITDB_GetCurrentField

Signature `char* ITDB_GetCurrentField(ITDB* db, int key, int field)`

Description Returns the specific field for the current iteration for `key`.

Argument `db` – the itdb on which to operate.
`key` – value of key for which to retrieve information.
`field` – field to retrieve.

Return Value String containing specified field.

6.1.15 ITDB_GetField

Signature
```
char* ITDB_GetField(ITDB* db, int key,
int field, int it)
```

Description Returns the specified field for the specified iteration for `key`.

Argument `db` – the itdb on which to operate.
`key` – key for which to retrieve information.
`field` – field to retrieve.
`it` – iteration from which to retrieve information

Return Value String containing specified field.

6.1.16 ITDB_GetFirstField

Signature
```
char* ITDB_GetFirstField(ITDB* db, int key,
int field, int it)
```

Description Returns the specified field for the earliest iteration that has data.

Argument `db` – the itdb on which to operate.
`key` – key for which to retrieve information.
`field` – field to retrieve.
`it` – iteration from which to retrieve information.

Return Value String containing specified field.

6.1.17 ITDB_GetLastField

Signature
```
char* ITDB_GetLastField(ITDB* db, int key, int field,
int it)
```

Description Returns the specified field for the latest iteration that has data.

Argument `db` – the itdb on which to operate.
`key` – key for which to retrieve information.
`field` – field to retrieve.
`it` – iteration from which to retrieve information.

Return Value String containing specified field.

6.1.18 ITDB_FieldNameToNumber

Signature `int ITDB_FieldNameToNumber(ITDB* db, char* name)`

Description Converts between field name and field number.

Argument `db` – the itdb on which to operate.
`name` – name to look up.

Return Value Number of the given field, or –1 if it was not found.

6.1.19 ITDB_FieldNumberToName

Signature `char* ITDB_FieldNumberToName(ITDB* db, int num)`

Description Converts between field number and field name.

Argument `db` – the itdb on which to operate.
`num` – number to look up.

Return Value String containing the field name, or `NULL` if it was not found.

6.1.20 ITDB_ItCreate

Signature `ITDB_It* ITDB_ItCreate(ITDB* db, int iteration)`

Description Creates an iterator for the records of the given iteration.

Argument `db` – database over which to iterate.
`iteration` – the number of the iteration over which to iterate.
If `iteration` is –1, then do all iterations.

Return Value An iterator set to the first record of the proper iteration.

6.1.21 ITDB_ItCreateRecord

Signature `ITDB_It* ITDB_ItCreateRecord(ITDB* db, int key)`

Description Creates an iterator for all iterations of the given record.

Argument `db` – database over which to iterate.
`key` – value of the key for which to return records.

Return Value An iterator set to the first record of the proper iteration.

6.1.22 ITDB_ItDestroy

Signature `void ITDB_ItDestroy(ITDB_It* it)`

Description Destroys an iterator and frees all resources.

Argument `it` – the iterator to destroy.

Return Value None.

6.1.23 ITDB_ItReset

Signature `void ITDB_ItReset(ITDB_It* it)`

Description Resets iterator to beginning.

Argument `it` – the iteration on which to operate.

Return Value None.

6.1.24 ITDB_ItAdvance

Signature `void ITDB_ItAdvance(ITDB_It* it)`

Description Advances to the next record.

Argument `it` – iteration which to operate.

Return Value None.

6.1.25 ITDB_ItMoreData

Signature `int ITDB_ItMoreData(ITDB_It* it)`

Description Is there more data?

Argument `it` – the iteration on which to operate.

Return Value 0 if there is no more data.
non-zero if there is more data.

6.1.26 ITDB_ItGetString

Signature `char* ITDB_ItGetString(ITDB_It* it)`

Description Returns the current record.

Argument `it` – the iteration on which to operate.

Return Value A null-terminated string containing a copy of the record. The caller is responsible for freeing this data.

6.1.27 ITDB_ItGetData

Signature `char* ITDB_ItGetData(ITDB_It* it)`

Description Returns the current record.

Argument `it` – the iteration on which to operate.

Return Value A pointer into the mmapped file. Changes to the string will change the actual file. This pointer should not be freed.

6.1.28 ITDB_StringToArray

Signature `char** ITDB_StringToArray(char* str)`

Description Converts a single string containing multiple fields to an array of strings containing single records.

Argument `str` – a string containing comma-separated fields.

Return Value An array of strings, one field per string. The last element of the array is `NULL`. The caller is responsible for freeing the returned pointer.

6.1.29 ITDB_ArrayToString

Signature `char* ITDB_ArrayToString(char** array)`

Description Convert an array of fields to a single string.

Argument `array` – an array of strings containing fields. The last element of the array must be set to `NULL`.

Return Value A single string containing the comma-separated fields.

6.2 Data Structures

6.2.1 ITDB

This structure contains all of the information about an iteration database.

```
typedef struct itdb_s
{
/** The current iteration number. **/
int iteration;

/** Used to construct the itdb filename. **/
char* base_filename;

/** Name of the current iteration file; base.#.it. **/
char* idx_filename;

/** File descriptor for current iteration file. **/
int it_fd;

/** Array of labels for the fields of the database. **/
char* field_labels;

/** The number of fields. **/
int num_fields;

/** End of the current iteration file. **/
size_t it_pos;

/** Index of all iteration files. **/
BTree* index;
} ITDB;
```

6.2.2 ITDB_It

This structure is an iterator into an iteration database.

```
typedef struct itdbit_s
{
/** The index for this iterator. **/
BTree* index;

/** The index iterator. **/
BTreeIt* index_it;

/** The iteration to iterate through.  –1 means all iterations.
**/
int iteration;

/** Iterate through one record only.  –1 means all records. **/
int key;

} ITDB_It;
```

7. SIMULATION OUTPUT

The simulation output subsystem has C structures and utility functions that are used to read and write TRANSIMS simulation output files.

7.1 Interface Functions

7.1.1 OutReadHeader

Signature `int` **`OutReadHeader`**`(FILE * file, TOutHeader * header)`

Description Read a header from an output table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.

Return Value Return nonzero if the header was successfully read, or zero if not.

7.1.2 OutWriteHeader

Signature
```
int OutWriteHeader (FILE * file,
const TOutHeader * header)
```

Description Write a header to an output table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.

Return Value Return nonzero if the header was successfully written, or zero if not.

7.1.3 OutSkipHeader

Signature `int` **`OutSkipHeader`** `(FILE * file)`

Description Skip a header from an output table.

Argument `file` – pointer to a `FILE` stream object.

Return Value Return nonzero if the header was successfully skipped, or zero if not.

7.1.4 OutHeaderHasField

Signature
```
int OutHeaderHasField (const TOutHeader * header,
const char * field)
```

Description Determine whether an output table header contains a specified field.

Argument `header` – pointer to an output table header structure.
`field` – pointer to a character string.

Return Value Return nonzero if the header contains the specified field, or zero if not.

7.1.5 OutReadNodeSpecification

Signature
```
int OutReadNodeSpecification (FILE * file,
TOutNodeSpecificationRecord * record)
```

Description Read a record from a node specification table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to an output node specification record structure.

Return Value Return nonzero if the record was successfully read, or zero if not.

7.1.6 OutWriteNodeSpecification

Signature
```
int OutWriteNodeSpecification (FILE * file, const
TOutNodeSpecificationRecord * record
```

Description Write a record to a node specification table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to an output node specification record structure.

Return Value Return nonzero if the record was successfully written, or zero if not.

7.1.7 OutReadLinkSpecification

Signature
```
int OutReadLinkSpecification (FILE * file,
TOutLinkSpecificationRecord * record)
```

Description Read a record from a link specification table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to an output link specification structure.

Return Value Return nonzero if the record was successfully read, or zero if not.

7.1.8 OutWriteLinkSpecification

Signature `int` **`OutWriteLinkSpecification`** `(FILE * const TOutLinkSpecificationRecord * record)`

Description Write a record to a link specification table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to an output link specification structure.

Return Value Return nonzero if the record was successfully written, or zero if not.

7.1.9 OutReadTravelerEventHeader

Signature `int` **`OutReadTravelerEventHeader`** `(FILE * file, TOutHeader * header, TOutTravelerEventRecord * record)`

Description Read a header from a traveler event table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a traveler event structure.

Return Value Return nonzero if the header was successfully read, or zero if not.

7.1.10 OutWriteTravelerEventHeader

Signature `int` **`OutWriteTravelEventHeader`** `(FILE * file, const TOutHeader * header, TOutTravelerEventRecord * record)`

Description Write a header to a traveler event table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a traveler event structure.

Return Value Return nonzero if the header was successfully written, or zero if not.

7.1.11 OutReadTravelerEvent

Signature `int` **`OutReadTravelerEvent`** `(FILE * file, TOutTravelerEventRecord * record)`

Description Read a record from a traveler event table.

Argument `file` – pointer to a `FILE` stream object.

`record` – pointer to a traveler event structure.

Return Value Return nonzero if the record was successfully read, or zero if not.

7.1.12 OutReadTravelerEventFromString

Signature
```
int OutReadTravelerEventFromString (const char * buf,
TOutTravelerEventRecord * record)
```

Description Read a record from a character buffer (which may not be null terminated).

Argument `buf` – pointer to a character string.
`record` – pointer to a traveler event structure.

Return Value Return nonzero if the record was successfully read, or zero if not.

7.1.13 OutWriteTravelerEvent

Signature
```
int OutWriteTravelerEvent (FILE * file,
const TOutTravelerEventRecord * record)
```

Description Write a record to a traveler event table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a traveler event structure.

Return Value Return nonzero if the record was successfully written, or zero if not.

7.1.14 OutReadVehicleEvolutionHeader

Signature
```
int OutReadVehicleEvolutionHeader (FILE * file,
TOutHeader * header,
TOutVehicleEvolutionRecord * record)
```

Description Read a header from a vehicle evolution table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a vehicle evolution structure.

Return Value Return nonzero if the header was successfully read, or zero if not.

7.1.15 OutWriteVehicleEvolutionHeader

Signature
```
int OutWriteVehicleEvolutionHeader (FILE * file,
```

```
const TOutHeader * header,
TOutVehicleEvolutionRecord * record)
```

Description Write a header to a vehicle evolution table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a vehicle evolution structure.

Return Value Return nonzero if the header was successfully written, or zero if not.

7.1.16 OutReadVehicleEvolution

Signature
```
int OutReadVehicleEvolution (FILE * file,
TOutVehicleEvolutionRecord * record)
```

Description Read a record from a vehicle evolution table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a vehicle evolution structure.

Return Value Return nonzero if the record was successfully read, or zero if not.

7.1.17 OutWriteVehicleEvolution

Signature
```
int OutWriteVehicleEvolution (FILE * file,
const TOutVehicleEvolutionRecord * record)
```

Description Write a record to a vehicle evolution table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a vehicle evolution structure.

Return Value Return nonzero if the record was successfully written, or zero if not.

7.1.18 OutReadIntersectionEvolutionHeader

Signature
```
int OutReadIntersectionEvolutionHeader (FILE * file,
TOutHeader * header,
TOutIntersectionEvolutionRecord * record)
```

Description Read a header from an intersection evolution table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to an intersection evolution structure.

Return Value Return nonzero if the header was successfully read, or zero if not.

7.1.19 OutWriteIntersectionEvolutionHeader

Signature
```
int OutWriteIntersectionEvolutionHeader (FILE * file,
const TOutHeader * header,
TOutIntersectionEvolutionRecord * record)
```

Description Write a header to an intersection evolution table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to an intersection evolution structure.

Return Value Return nonzero if the header was successfully written, or zero if not.

7.1.20 OutReadIntersectionEvolution

Signature
```
int OutReadIntersectionEvolution (FILE * file,
TOutIntersectionEvolutionRecord * record)
```

Description Read a record from an intersection evolution table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to an intersection evolution structure.

Return Value Return nonzero if the record was successfully read, or zero if not.

7.1.21 OutWriteIntersectionEvolution

Signature
```
int OutWriteIntersectionEvolution (FILE * file,
const TOutIntersectionEvolutionRecord * record)
```

Description Write a record to an intersection evolution table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to an intersection evolution structure

Return Value Return nonzero if the record was successfully written, or zero if not.

7.1.22 OutReadTrafficControlEvolutionHeader

Signature
```
int OutReadTrafficControlEvolutionHeader (FILE * file,
TOutHeader * header,
TOutTrafficControlEvolutionRecord * record)
```

Description Read a header from a traffic control evolution table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a traffic control evolution structure.

Return Value Return nonzero if the header was successfully read, or zero if not.

7.1.23 OutWriteTrafficControlEvolutionHeader

Signature
```
int OutWriteTrafficControlEvolutionHeader (FILE * file,
const TOutHeader * header,
TOutTrafficControlEvolutionRecord * record)
```

Description Write a header to a traffic control evolution table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a traffic control evolution structure.

Return Value Return nonzero if the header was successfully written, or zero if not.

7.1.24 OutReadTrafficControlEvolution

Signature
```
int OutReadTrafficControlEvolution (FILE * file,
TOutTrafficControlEvolutionRecord * record)
```

Description Read a record from a traffic control evolution table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a traffic control evolution structure.

Return Value Return nonzero if the record was successfully read, or zero if not.

7.1.25 OutWriteTrafficControlEvolution

Signature
```
int OutWriteTrafficControlEvolution (FILE * file,
const TOutTrafficControlEvolutionRecord * record)
```

Description Write a record to a traffic control evolution table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a traffic control evolution structure.

Return Value Return nonzero if the record was successfully written, or zero if not.

7.1.26 OutReadLinkTimeSummaryHeader

Signature
```
int OutReadLinkTimeSummaryHeader (FILE * file,
TOutHeader * header, TOutLinkTimeSummaryRecord * record)
```

Description Read a header from a link time summary table.

Argument
`file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a link time summary structure.

Return Value Return nonzero if the header was successfully read, or zero if not.

7.1.27 OutWriteLinkTimeSummaryHeader

Signature
```
int OutWriteLinkTimeSummaryHeader (FILE * file,
const TOutHeader * header,
TOutLinkTimeSummaryRecord * record)
```

Description Write a header to a link time summary table.

Argument
`file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a link time summary structure.

Return Value Return nonzero if the header was successfully written, or zero if not.

7.1.28 OutReadLinkTimeSummary

Signature
```
int OutReadLinkTimeSummary (FILE * file,
TOutLinkTimeSummaryRecord * record)
```

Description Read a record from a link time summary table.

Argument
`file` – pointer to a `FILE` stream object.
`record` – pointer to a link time summary structure.

Return Value Return nonzero if the record was successfully read, or zero if not.

7.1.29 OutWriteLinkTimeSummary

Signature
```
int OutWriteLinkTimeSummary (FILE * file,
const TOutLinkTimeSummaryRecord * record)
```

Description Write a record to a link time summary table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a link time summary structure.

Return Value Return nonzero if the record was successfully written, or zero if not.

7.1.30 OutReadLinkSpaceSummaryHeader

Signature
```
int OutReadLinkSpaceSummaryHeader (FILE * file,
TOutHeader * header,
TOutLinkSpaceSummaryRecord * record)
```

Description Read a header from a link space summary table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a link space summary structure.

Return Value Return nonzero if the header was successfully read, or zero if not.

7.1.31 OutWriteLinkSpaceSummaryHeader

Signature
```
int OutWriteLinkSpaceSummaryHeader (FILE * file,
const TOutHeader * header,
TOutLinkSpaceSummaryRecord * record)
```

Description Write a header to a link space summary table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a link space summary structure.

Return Value Return nonzero if the header was successfully written, or zero if not.

7.1.32 OutReadLinkSpaceSummary

Signature
```
int OutReadLinkSpaceSummary (FILE * file,
TOutLinkSpaceSummaryRecord * record)
```

Description Read a record from a link space summary table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a link space summary structure.

Return Value Return nonzero if the record was successfully read, or zero if not.

7.1.33 OutWriteLinkSpaceSummary

Signature
```
int OutWriteLinkSpaceSummary (FILE * file,
const TOutLinkSpaceSummaryRecord * record)
```

Description Write a record to a link space summary table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a link space summary structure.

Return Value Return nonzero if the record was successfully written, or zero if not.

7.1.34 OutReadLinkVelocitySummaryHeader

Signature
```
int OutReadLinkVelocitySummaryHeader (FILE * file,
TOutHeader * header,
TOutLinkVelocitySummaryRecord * record)
```

Description Read a header from a link velocity summary table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table structure defined.
`TOutLinkVelocitySummaryRecord` – pointer to a link velocity summary structure.

Return Value Return nonzero if the header was successfully read, or zero if not.

7.1.35 OutWriteLinkVelocitySummaryHeader

Signature
```
int OutWriteLinkVelocitySummaryHeader (FILE * file,
const TOutHeader * header,
TOutLinkVelocitySummaryRecord * record)
```

Description Write a header to a link velocity summary table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a link velocity summary structure.

Return Value Return nonzero if the header was successfully written, or zero if not.

7.1.36 OutReadLinkVelocitySummary

Signature
```
int OutReadLinkVelocitySummary (FILE * file,
TOutLinkVelocitySummaryRecord * record)
```

Description Read a record to a link velocity summary table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a link velocity summary structure.

Return Value Return nonzero if the record was successfully read, or zero if not.

7.1.37 OutWriteLinkVelocitySummary

Signature
```
int OutWriteLinkVelocitySummary (FILE * file,
const TOutLinkVelocitySummaryRecord * record)
```

Description Write a record to a link velocity summary table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a link velocity summary structure.

Return Value Return nonzero if the record was successfully written, or zero if not.

7.1.38 OutReadLinkEnergySummaryHeader

Signature
```
int OutReadLinkEnergySummaryHeader (FILE * file,
TOutHeader * header,
TOutLinkEnergySummaryRecord * record)
```

Description Read a header to a link energy summary table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a link energy summary structure.

Return Value Return nonzero if the header was successfully read, or zero if not.

7.1.39 OutWriteLinkEnergySummaryHeader

Signature
```
int OutWriteLinkEnergySummaryHeader (FILE * file,
const TOutHeader *header,
TOutLinkEnergySummaryRecord * record)
```

Description Write a header to a link energy summary table.

Argument `file` – pointer to a `FILE` stream object.
`header` – pointer to an output table header structure.
`record` – pointer to a link energy summary structure.

Return Value Return nonzero if the header was successfully written, or zero if not.

7.1.40 OutReadLinkEnergySummary

Signature `int` **`OutReadLinkEnergySummary`** `(FILE * file, TOutLinkEnergySummaryRecord * record)`

Description Read a record to a link energy summary table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a link energy summary structure.

Return Value Return nonzero if the record was successfully read, or zero if not.

7.1.41 OutWriteLinkEnergySummary

Signature `int` **`OutWriteLinkEnergySummary`** `(FILE * file, const TOutLinkEnergySummaryRecord * record)`

Description Write a record to a link energy summary table.

Argument `file` – pointer to a `FILE` stream object.
`record` – pointer to a link energy summary structure.

Return Value Return nonzero if the record was successfully written, or zero if not.

7.1.42 CreativeEventFileIndex

Signature `void` **`CreateEventFileIndex`** `(const char * filename)`

Description Create two indexes for a traveler event file. The first index (with extension *.trv.idx*) is sorted by traveler ID as the primary key and trip ID as the secondary key. The second index (with extension *.loc.idx*) is sorted by location ID as the primary key and traveler ID as the secondary key.

Argument `filename` – pointer to a character string containing the name of a traveler event file.

Return Value No value returned.

7.2 Data Structures

7.2.1 TOutHeader

This structure is used for the output table header.

```
typedef struct
{
/** The field names. **/
INT8 fFields[512];

} TOutHeader;
```

7.2.2 TOutNodeSpecificationRecord

This structure is used for output node specification table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

} TOutNodeSpecificationRecord;
```

7.2.3 TOutLinkSpecificationRecord

This structure is used for output link specification table records.

```
typedef struct
{
/** The LINK field. **/
INT32 fLink;

} TOutLinkSpecificationRecord;
```

7.2.4 TOutTravelerEventRecord

This structure is used for traveler event records.

```
typedef structure
{
/** The TIME field. **/
REAL64 fTime;

/** The TRAVELER field.  **/
INT32 fTraveler;

/** The TRIP field. **/
```

```
INT32 fTrip;

/** The LEG field. **/
INT32 fLeg;

/** The VEHICLE field. **/
INT32 fVehicle;

/** The VEHTYPE field. **/
INT32 fVehtype;

/** The VSUBTYPE field. **/
INT32 fVsubtype;

/** The ROUTE field. **/
INT32 fRoute;

/** The STOPS field. **/
INT32 fStops;

/** The YIELDS field. **/
INT32 fYields;

/** The SIGNALS field. **/
INT32 fSignals;

/** The TURN field. **/
INT32 fTurn;

/** The STOPPED field. **/
REAL64 fStopped;

/** the ACCELS field. **/
REAL64 fAccels;

/** The TIMESUM field. **/
REAL64 fTimesum;

/** The DISTANCESUM field. **/
REAL64 fDistancesum;

/** The USER field. **/
INT32 fUser;

/** The Anomaly field. **/
INT32 fAnomaly;

/** The STATUS field. **/
INT32 fStatus;

/** The LINK field. **/
INT32 fLink;
```

```
/** The NODE field. **/
INT32 fNode;

/** The LOCATION field. **/
INT32 fLocation;

/** Private:  The i/o formats. **/
INT8 fFormat[2] [93];

/** Private: The pointers to the data. **/
INT32 fOffsets[22];

} TOutTravelerEventRecord;
```

7.2.5 TOutVehicleEvolutionRecord

This structure is used for vehicle evolution records.

```
typedef struct
{
/** The TIME field. **/
REAL64 fTime;

/** The DRIVER field. **/
INT32 fDriver;

/** The VEHICLE field. **/
INT32 fVehicle;

/** The VEHTYPE field. **/
INT32 fVehtype;

/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
INT32 fNode.;

/** The LANE field. **/
INT32 fLane;

/** The DISTANCE field. **/
REAL64 fDistance;

/** The VELOCITY field. **/
REAL64 fVelocity;

/** The ACCELER field. **/
REAL64 fAcceler;
```

```
/** The PASSENGERS field. **/
INT32 fPassengers;

/** The EASTING field. **/
REAL64 fEasting;

/** The NORTHING field. **/
REAL64 fNorthing;

/** The ELEVATION field. **/
REAL64 fElevation;

/** The AZIMUTH field. **/
REAL64 fAzimuth;

/** The USER field. **/
INT32 fUser;

/** Private: The i/o formats. **/
INT8 fFormat[2] [72];

/** Private: The pointers to the data. **/
INT32 fOffsets[16];

} TOutVehicleEvolutionRecord;
```

7.2.6 TOutIntersectionEvolutionRecord

This structure is used for intersection evolution records.

```
typedef struct
{
/** The TIME field. **/
REAL64 fTime;

/** The VEHICLE field. **/
INT32 fVehicle;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The LANE field. **/
INT32 fLane;

/** The QINDEX field. **/
INT32 fQindex;

/** Private: The i/o formats. **/
```

```
INT8 fFormat[2] [25];

/** Private:  The pointer to the data. **/
INT32 fOffsets [6];

} TOutIntersectionEvolutionRecord;
```

7.2.7 TOutTrafficControlEvolutionRecord;

This structure is used for traffic control evolution records.

```
typedef struct
{
/** The TIME field. **/
REAL64 fTime;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The LANE field. **/
INT32 fLane;

/** The SIGNAL field. **/
INT32 fSignal;

/** Private: The i/o formats. **/
INT8 fFormat [2] [21];

/** Private: The pointers to the data. **/
INT32 fOffsets[5];

} TOutTrafficControlEvolutionRecord;
```

7.2.8 TOutLinkTimeSummaryRecord

This structure is used for link time summary records.

```
typedef struct
{
/** The TIME field. **/
REAL64 fTime;

/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
INT32 fNode;
```

```
/** The LANE field. **/
INT32 fLane;

/** The TURN field. **/
INT32 fTurn;

/** The COUNT field. **/
INT32 fCount;

/** The SUM field. **/
REAL64 fSum;

/** The SUMSQUARES field. **/
REAL64 fSumsquares;

/** The VCOUNT field. **/
INT32 fVCount;

/** The VSUM field. **/
REAL64 fVSum;

/** The VSUMSQUARES field. **/
REAL64 fVSumsquares;

/** Private: The i/o formats. **/
INT8 fFormat[2] [49];

/** Private: The pointers to the data. **/
INT32 fOffsets[11];

} TOutLinkTimeSummaryRecord;
```

7.2.9 TOutLinkSpaceSummaryRecord

This structure is used for link space summary records.

```
typedef struct
{
/** The TIME field. **/
REAL64 fTime;

/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
INT32 fNode;

/** The LANE field. **/
INT32 fLane;

/** The DISTANCE field. **/
```

```
REAL64 fDistance;

/** The COUNT field. **/
INT32 fCount;

/** The SUM field. **/
REAL64 fSum;

/** The SUMSQUARES field. **/
REAL64 fSumsquares;

/** Private: The i/o formats. **/
INT8 fFormat[2] [36];

/** Private: The pointers to the data. **/
INT32 fOffsets[8];

} TOutLinkSpaceSummaryRecord;
```

7.2.10 TOutLinkVelocitySummaryRecord

This structure is used for link velocity summary records.

```
/** Maximum allowed number of bins in a histogram. **/
#define HISTOGRAM_MAX_BINS 100

/** Structure for link velocity summary records. **/
typedef struct
{

/** The TIME field. **/
REAL64 fTime;

/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
INT32 fNode.

/** The DISTANCE field. **/
READ64 fDistance;

/** The COUNT fields. **/
INT32 fCount [HISTOGRAM_MAX_BINS];

/** The number of bins in the histogram. **/
INT32 fNumberBins;

/** Private: The i/o formats. **/
INT8 fFormat[2] [18 + 4 * HISTOGRAM_MAX_BINS];
```

```
/** Private:  The pointers to the data. **/
INT32 fOffsets[4 + HISTOGRAM_MAX_BINS];

} TOutLinkVelocitySummaryRecord;
```

7.2.11 TOutLinkEnergySummaryRecord

This structure is used for link energy summary records.

```
/** Maximum allowed number of bins in a histogram. **/
#define HISTOGRAM_MAX_BINS_100

/** Structure for link energy summary records. **/
typedef struct
{
/** The TIME field. **/
REAL64 fTime;

/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
INT32 fNode;

/** The ENERGY fields. **/
INT32 fEnergy[HISTOGRAM_MAX_BINS];

/** The number of bins in the histogram. **/
INT32 fNumberBins;

/** Private: The i/o formats. **/
INT8 fFormat[2] [13 + 3 * HISTOGRAM_MAX_BINS];

/** Private: The pointers to the data. **/
INT32 fOffsets[3 + HISTOGRAM_MAX_BINS];

} TOutLinkEnergySummaryRecord;
```

8. TRANSIT

The transit subsystem has C structures and utility functions used to read and write data from a TRANSIMS transit route file, a TRANSIMS transit schedule file, or a TRANSIMS transit zone file.

8.1 Interface Functions

The function **`getNextTransitRouteData()`** reads transit route data from a transit route file in ASCII format. The function stores the information in an unmodifiable data structure (`TransitRouteData`) and returns a pointer to the structure. Because the calling program cannot modify the `TransitRouteData` structure, the data should be copied if it needs to be changed.

The function **`writeTransitRouteData()`** takes a `TransitRouteData` structure as an argument containing the information to be written.

All of these functions work similarly:

- **`getNextTransitScheduleData() ,`**
- **`writeTransitScheduleData () ,`**
- **`getNextZoneData() , and`**
- **`writeTransitZoneData().`**

The **`getNextTransitRouteData ()`**, **`getNextTransitScheduleData ()`**, or **`getNextTransitZoneData ()`** function, when combined with the **`moreTransit ()`** function, provide a mechanism for iterating through the appropriate transit file.

8.1.1 moreTransitData

Signature `int` **`moreTransitData`**`(FILE * const)`

Description Boolean function used to control iteration through the transit file.

Argument `fp – FILE *` for the transit file, which must be open for reading.

Return Value 1 if not at end of transit data file.
0 if EOF has been reached.

8.1.2 getNextTransitRouteData

Signature
```
const TransitRouteDate * getNextTransitRouteData
(FILE * const fp)
```

Description Reads transit route data from the transit route file.

Argument `fp – FILE *` to the transit route file, which must be open for reading.

Return Value The address (containing the transit route read from the file) of a `TransitRouteData` structure.
Returns `NULL` on error.

8.1.3 writeTransitRouteData

Signature
```
int writeTransitRouteData
(FILE * const fp, TransitRouteData * data)
```

Description Writes the `TransitRouteData` into a line of the given transit route file.

Argument `fp` – `FILE *` to the transit route file, which must be open for reading.
`data` – address of a `TransitRouteData` structure containing the data to be written.

Return Value 1 on success.
0 on error.

8.1.4 getNextTransitScheduleData

Signature
```
const TransitScheduleData * getNextTransitScheudleData
(FILE * const fp)
```

Description Reads transit schedule data from the transit schedule file.

Argument `fp – FILE *` to the transit schedule file, which must be open for reading.

Return Value The address (containing the transit schedule read from the file) of a `TransitScheduleData` structure.
Returns `NULL` on error.

8.1.5 writeTransitScheduleData

Signature
```
int writeTransitScheduleData
(FILE * const fp, TransitScheduleData * data)
```

Description Writes the given `TransitScheduleData` into a line of the given transit schedule file.

Argument `fp` – `FILE *` to the transit schedule file, which must be open for writing.
`data` – address of a `TransitScheduleData` structure containing the

data to be written.

Return Value 1 on success.
0 on error.

8.1.6 getNextTransitZoneData

Signature
```
const TransitZoneData * getNextTransitZoneData
(FILE * const fp)
```

Description Reads transit zone data from the transit zone file.

Argument `fp – FILE *` to the transit zone file, which must be open for reading.

Return Value The address of a `TransitZoneData` structure containing the transit zone read from the file.
Returns `NULL` on error.

8.1.7 writeTransitZoneData

Signature
```
int writeTransitZoneData(FILE * const fp,
TransitZoneData * data)
```

Description Writes the given `TransitZoneData` into a line of the given transit route file.

Argument `fp` – `FILE *` to the transit zone file, which must be open for writing.
`data` – address of a `TransitZoneData` structure containing the data to be written.

Return Value 1 on success.
0 on error.

8.2 Data Structures

8.2.1 TransitStopData

This structure is used for transit stop data as specified in the transit route file.

```
typedef struct transitstopdata_s
{
/** The stop Id. **/
INT32 fStopId;

/** /The link Id. **/
INT32 fLinkId;
```

```
/** The node ID. **/
INT32 fNodeId;

/** The transit zone (0 if none) **/
INT32 fTransitZone;

} TransitStopData;
```

8.2.2 TransitRouteData

This structure is used for transit route data as specified in the transit route file.

```
typedef struct transitroutedata s
{

/** The route Id. **/
INT32 fRouteId;

/** The number of stops. **/
INT32 fNumStops;

/** The type of transit for this route. **/
INT8 fTransitType[16];

/** An array of info about stops for this route **/
TransitStopData *fStops;

} TransitRouteData;
```

8.2.3 TransitScheduleData

This structure is used for transit schedule data as specified in the transit schedule file.

```
typedef struct transitscheduledata s
{

/** The stop Id. **/
INT32 fStopId;

/** The route Id. **/
INT32 fRouteId;

/** The arrival time. **/
INT32 fArrivalTime;

} TransitscheduleData;
```

8.2.4 TransitZoneData

This structure is used for transit zone data as specified in the transit zone file.

```
typedef struct transitzonedata s
{

/** The source zone. **/
INT32 fFromZone;

/** The destination zone. **/
INT32 fToZone;

/** The cost of travel from FromZone to ToZone on TransitType, in
cents. **/
INT32 fCost;

/** The type of transit for this route. **/
INT8 fTransitType[16];

} TransitZoneData;
```

9. NETWORK

9.1 Interface Functions

The following section deals with the network subsystem, which has C structures and utility functions for reading and writing network data files.

9.1.1 NetReadHeader

Signature `int NetReadHeader(FILE * file, TNetHeader * header)`

Description Read a header from a network table.

Argument `file` – `FILE` pointer for the network data table.
`header` – pointer to TNetHeader structure into which the header is read.

Return Value Return nonzero if the header was successfully read, or zero if not.

9.1.2 NetWriteHeader

Signature
```
int NetWriteHeader(FILE * file,
const TNetHeader * header)
```

Description Write a header from a network table.

Argument `file` – `FILE` pointer for the network data table.
`header` – pointer to `TNetHeader` structure from which the header is written.

Return Value Return nonzero if the header was successfully written, or zero if not.

9.1.3 NetSkipHeader

Signature `int NetSkipHeader(FILE * file)`

Description Skip a header from a network table.

Argument `file` – `FILE` pointer for the network data table.

Return Value Return nonzero if the header was successfully skipped, or zero if not.

9.1.4 NetReadActivityLocationHeader

Signature `int` **`NetReadActivityLocationHeader`**`(FILE* file, TNetHeader* header, TNetActivityLocationRecord* record)`

Description Read a header from an activity location table.

Argument `file` – `FILE` pointer for the network data table.
`header` – pointer to `TNetHeader` structure into which the header is read.
`record` – pointer to `TNetActivityLocationRecord` structure which is initialized based on the header contents.

Return Value Return nonzero if the header was successfully read, or zero if not.

9.1.5 NetReadNode

Signature `int` **`NetReadNode`**`(FILE * file, TNetNodeRecord * record)`

Description Read a record from a node table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetNodeRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.6 NetWriteNode

Signature `int` **`NetWriteNode`**`(FILE * file, const TNetNodeRecord * record)`

Description Write a record to a node table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetNodeRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.7 NetReadLink

Signature `int` **`NetReadLink`**`(FILE * file, TNetLinkRecord * record)`

Description Read a record from a link table.

Argument `file` – `FILE` pointer for the network data table
`record` – pointer to `TNetLinkRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.8 NetWriteLink

Signature
```
int NetWriteLink(FILE * file,
const TNetLinkRecord * record)
```

Description Write a record to a link table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetLinkRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.9 NetReadSpeed

Signature
```
int NetReadSpeed(FILE * file, TNetSpeedRecord * record)
```

Description Read a record from a speed table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSpeedRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.10 NetWriteSpeed

Signature
```
int NetWriteSpeed(FILE * file,
const TNetSpeedRecord * record)
```

Description Write a record to a speed table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSpeedRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.11 NetReadPocket

Signature
```
int NetReadPocket(FILE * file,
TNetPocketRecord * record)
```

Description Read a record from a pocket lane table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetPocketRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.12 NetWritePocket

Signature
```
int NetWritePocket(FILE * file,
const TNetPocketRecord * record)
```

Description Write a record to a pocket lane table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetPocketRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.13 NetReadLaneUse

Signature
```
int NetReadLaneUse(FILE * file,
TNetLaneUseRecord * record)
```

Description Read a record from a lane use table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetLaneUseRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.14 NetWriteLaneUse

Signature
```
int NetWriteLaneUse(FILE * file,
const TNetLaneUseRecord * record)
```

Description Write a record to a lane use table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetLaneUseRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.15 NetReadParking

Signature
```
int NetReadParking(FILE * file,
TNetParkingRecord * record)
```

Description Read a record from a parking table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetParkingRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.16 NetWriteParking

Signature
```
int NetWriteParking(FILE * file,
const TNetParkingRecord * record)
```

Description Write a record to a parking table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetParkingRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.17 NetReadBarrier

Signature
```
int NetReadBarrier(FILE * file,
TNetBarrierRecord * record)
```

Description Read a record from a barrier table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetBarrierRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.18 NetWriteBarrier

Signature
```
int NetWriteBarrier(FILE * file,
const TNetBarrierRecord * record)
```

Description Write a record to a barrier table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetBarrierRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.19 NetReadTransitStop

Signature
```
int NetReadTransitStop(FILE * file,
TNetTransitStopRecord* record)
```

Description Read a record from a transit stop table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetTransitStopRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.20 NetWriteTransitStop

Signature
```
int NetWriteTransitStop(FILE * file,
const TNetTransitStopRecord * record)
```

Description Write a record to a transit stop table.

Argument `file – FILE` pointer for the network data table.
`record –` pointer to `TNetTransitStopRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.21 NetReadLaneConnectivity

Signature
```
int NetReadLaneConnectivity(FILE * file,
TNetLaneConnectivityRecord * record)
```

Description Read a record from a lane connectivity table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetLaneConnectivityRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.22 NetWriteLaneConnectivity

Signature
```
int NetWriteLaneConnectivity(FILE * file,
const TNetLaneConnectivityRecord * record)
```

Description Write a record to a lane connectivity table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetLaneConnectivityRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.23 NetReadTurnProhibition

Signature
```
int NetReadTurnProhibition(FILE * file,
TNetTurnProhibitionRecord * record)
```

Description Read a record from a turn prohibition table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetTurnProhibitionRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.24 NetWriteTurnProhibition

Signature
```
int NetWriteTurnProhibition(FILE * file,
const TNetTurnProhibitionRecord * record)
```

Description Write a record to a turn prohibition table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetTurnProhibitionRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.25 NetReadUnsignalizedNode

Signature
```
int NetReadUnsignalizedNode(FILE * file,
TNetUnsignalizedNodeRecord * record)
```

Description Read a record from an unsignalized node table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetUnsignalizedNodeRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.26 NetWriteUnsignalizedNode

Signature
```
int NetWriteSignalizedNode(FILE * file,
const TNetUnsignalizedNodeRecord * record)
```

Description Write a record to an unsignalized node table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetUnsignalizedNodeRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.27 NetReadSignalizedNode

Signature
```
int NetReadSignalizedNode(FILE * file,
TNetSignalizedNodeRecord * record)
```

Description Read a record from a signalized node table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSignalizedNodeRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.28 NetWriteSignalizedNode

Signature
```
int NetWriteSignalizedNode(FILE * file,
const TNetSignalizedNodeRecord * record)
```

Description Write a record to a signalized node table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSignalizedNodeRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.29 NetReadPhasingPlan

Signature
```
int NetReadPhasingPlan(FILE * file,
TNetPhasingPlanRecord * record)
```

Description Read a record from a phasing plan table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetPhasingPlanRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.30 NetWritePhasingPlan

Signature
```
int NetWritePhasingPlan(FILE * file,
const TNetPhasingPlanRecord * record)
```

Description Write a record to a phasing plan table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetPhasingPlanRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.31 NetReadTimingPlan

Signature
```
int NetReadTimingPlan(FILE * file,
TNetTimingPlanRecord * record)
```

Description Read a record from a timing plan table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetTimingPlanRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.32 NetWriteTimingPlan

Signature
```
int NetWriteTimingPlan(FILE * file,
const TNetTimingPlanRecord * record)
```

Description Write a record to a timing plan table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetTimingPlanRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.33 NetReadDetector

Signature
```
int NetReadDetector(FILE * file,
TNetDetectorRecord * record)
```

Description Read a record from a detector table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetDetectorRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.34 NetWriteDetector

Signature
```
int NetWriteDetector(FILE * file,
const TNetDetectorRecord * record)
```

Description Write a record to a detector table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetDetectorRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.35 NetReadSignalCoordinator

Signature
```
int NetReadSignalCoordinator(FILE * file,
TNetSignalCoordinatorRecord * record)
```

Description Read a record from a signal coordinator table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSignalCoordinatorRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.36 NetWriteSignalCoordinator

Signature
```
int NetWriteSignalCoordinator(FILE * file,
const TNetSignalCoordinatorRecord * record)
```

Description Write a record to a signal coordinator table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetSignalCoordinatorRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.37 NetReadActivityLocation

Signature
```
int NetReadActivityLocation(FILE * file,
TNetActivityLocationRecord * record)
```

Description Read a record from an activity location table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetActivityLocationRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.38 NetWriteActivityLocation

Signature
```
int NetWriteActivityLocation(FILE * file, const
TNetActivityLocationRecord * record)
```

Description Write a record to a process link table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetActivityLocationRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.39 NetReadProcessLink

Signature
```
int NetReadProcessLink(FILE * file,
TNetProcessLinkRecord * record)
```

Description Read a record from a process link table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetProcessLinkRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.40 NetWriteProcessLink

Signature
```
int NetWriteProcessLink(FILE * file,
const TNetProcessLinkRecord * record)
```

Description Write a record to a process link table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetProcessLinkRecord` structure from which the record is written.

Return Value Return nonzero if the record was successfully written, or zero if not.

9.1.41 NetReadStudyAreaLink

Signature
```
int NetReadStudyAreaLink(FILE * file,
TNetStudyAreaLinkRecord * record)
```

Description Read a record from a study area link table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetStudyAreaLinkRecord` structure into which the record is read.

Return Value Return nonzero if the record was successfully read, or zero if not.

9.1.42 NetWriteStudyAreaLink

Signature
```
int NetWriteStudyAreaLink(FILE * file,
const TNetStudyAreaLinkRecord * record)
```

Description Write a record to a study area link table.

Argument `file` – `FILE` pointer for the network data table.
`record` – pointer to `TNetStudyAreaLinkRecord` structure from which the record is written.

Return Value

Return nonzero if the record was successfully written, or zero if not.

9.2 Data Structures

9.2.1 TNetHeader

This structure is used for all of the network table header.

```
typedef struct
{
/** The field names. **/
INT8 fFields[512];

} TNetHeader;
```

9.2.2 TNetNodeRecord

This structure is used for network node table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The EASTING field. **/
REAL64 fEasting;

/** The NORTHING field.  **/
REAL64 fNorthing;

/** The ELEVATION field. **/
REAL64 fElevation;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetNodeRecord;
```

9.2.3 TNetLinkRecord

This structure isused for network link table records.

```
typedef struct
{
```

```
/** The ID field. **/
INT32 fId;

/** The NAME field. **/
INT8 fName[51];

/** The NODEA field. **/
INT32 fNodea;

/** The NODEB field. **/
INT32 fNodeb;

/** The PERMLANESA field. **/
INT32 fPermlanesa;

/** The PERMLANESB field. **/
INT32 fPermlanesb;

/** The LEFTPCKTSA field. **/
INT32 fLeftpcktsa;

/** The LEFTPCKTSB field.  **/
INT32 fLeftpcktsb;

/** The RIGHTPCKTSA field. **/
INT32 fRightpcktsa;

/** The RIGHTPCKTSB field. **/
INT32 fRightpcktsb;

/** The TWOWAYTURN field. **/
INT8 fTwowayturn[2];

/** The LENGTH field. **/
REAL64 fLength;

/** The GRADE field. **/
REAL64 fGrade;

/** The SETBACKA field. **/
REAL64 fSetbacka;

/** The SETBACKB field. **/
REAL64 fSetbackb;

/** The CAPACITYA field. **/
INT32 fCapacitya;

/** The CAPACITYB field. **/
INT32 fCapacityb;

/** The SPEEDLMTA field. **/
```

```
REAL64 fSpeedlmta;

/** The SPEEDLMTB field. **/
REAL64 fSpeedlmtb;

/** The FREESPDA field. **/
REAL64 fFreespda;

/**  The FREESPDB field. **/
REAL64 fFreespdb;

/** The FUNCTCLASS field. **/
INT8 fFunctclass[11];

/** The THRUA field. **/
INT32 fThrua;

/** The THRUB field. **/
INT32 fThrub;

/** The COLOR field. **/
INT32 fColor;

/** The VEHICLE field. **/
INT8 fVehicle[101];

/** the NOTES field. **/
INT8 fNotes[256];

} TNetLinkRecord;
```

9.2.4 TNetSpeedRecord

This structure is used for network speed table records.

```
typedef struct
{
/** The LINK field. **/
INT32 fLink;

/** The NODE field. **/
INT32 fNode;

/** The SPEEDLMT field. **/
REAL64 fSpeedlmt;

/**  The FREESPD field. **/
REAL64 fFreespd;

/** The VEHICLE field. **/
INT8 fVehicle[101];
```

```
/** The STARTTIME field. **/
INT8 fStarttime[9];

/** The ENDTIME field. **/
INT8 fEndtime[9];

/** The NOTES field. **/
INT8 fNotes[256]

} TNetSpeedRecord;
```

9.2.5 TNetPocketRecord

This structure is used for network pocket lane table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The OFFSET field. **/
REAL64 fOffset;

/** The LANE field. **/
INT32 fLane;

/** The STYLE field. **/
INT8 fStyle[2];

/** The LENGTH field. **/
REAL64 fLength;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetPocketRecord;
```

9.2.6 TNetLaneUseRecord

This structure is used for network lane use table records.

```
typedef struct
{
/** The NODE field. **/
```

```
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The LANE field. **/
INT32 fLane;

/** The VEHICLE field. **/
INT8 fVehicle[101];

/** The RESTRICT field. **/
INT8 fRestrict[2];

/** The STARTTIME field. **/
INT8 fStarttime[9];

/** The ENDTIME field. **/
INT8 fEndtime[9];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetLaneUseRecord
```

9.2.7 TNetParkingRecord

This structure is used for network parking table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The OFFSET field. **/
REAL64 fOffset;

/** The STYLE field. **/
INT8 fStyle[6];

/** The CAPACITY field. **/
INT32 fCapacity;

/** The GENERIC field. **/
INT8 fGeneric[2];
```

```
/** The VEHICLE field. **/
INT8 fVehicle[101];

/** The STARTTIME field. **/
INT8 fStarttime[9];

/** The ENDTIME field. **/
INT8 fEndtime[9];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetParkingRecord;
```

9.2.8 TNetBarrierRecord

This structure is used for network barrier table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The OFFSET field. **/
REAL64 fOffset;

/** The LANE field. **/
INT32 fLane;

/** The STYLE field. **/
INT8 fStyle[11];

/** The LENGTH field. **/
REAL64 fLength;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetBarrierRecord;
```

9.2.9 TNetTransitStopRecord

This structure is used for network transit stop table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The NAME field. **/
INT8 fName[51];

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The OFFSET field. **/
REAL64 fOffset;

/** The VEHICLE field. **/
INT8 fVehicle[101];

/** The STYLE field. **/
INT8 fStyle[11];

/** The CAPACITY field. **/
INT32 fCapacity;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetTransitStopRecord;
```

9.2.10 TNetLaneConnectivityRecord

This structure is used for network lane connectivity table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

/** The INLINK field. **/
INT32 fInlink;

/** The INLANE field. **/
INT32 fInlane;
```

```
/** The OUTLINK field. **/
INT32 fOutlink;

/** The OUTLANE field. **/
INT32 fOutlane;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetLaneConnectivityRecord;
```

9.2.11 TNetTurnProhibitionRecord

This structure is used for network turn prohibition table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

/** The INLINK field. **/
INT32 fInlink;

/** The OUTLINK field. **/
INT32 fOutlink;

/** The STARTTIME field. **/
INT8 fStarttime[9];

/** The ENDTIME field. **/
INT8 fEndtime[9];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetTurnProhibitionRecord;
```

9.2.12 TNetUnsignalizedNodeRecord

This structure is used for network unsignalized node table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

/**  The INLINK field. **/
INT32 fInlink;

/** The SIGN field. **/
INT8 fSign[2];
```

```
/** The NOTES field. **/
INT8 fNotes;

} TNetUnsignalizedNodeRecord;
```

9.2.13 TNetSignalizedNodeRecord

This structure is used for network signalized node table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

/** The TYPE field. **/
INT8 fType[2];

/** The PLAN field. **/
INT32 fPlan;

/** The OFFSET field. **/
REAL64 fOffset;

/** The STARTTIME field. **/
INT8 fStarttime[9];

/** The COORDINATR field. **/
INT32 fCoordinatr;

/** The RING field. **/
INT8  fRing[2];

/** The ALGORITHM field. **/
INT8 fAlgorithm[11];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetSignalizedNodeRecord;
```

9.2.14 TNetPhasingPlanRecord

This structure is used for network phasing plan table records.

```
typedef struct
{
/** The NODE field. **/
INT32 fNode;

/** The PLAN field. **/
```

```
INT32 fPlan;

/** The PHASE field. **/
INT32 fPhase;

/** The INLINK field. **/
INT32 fInlink;

/** The OUTLINK field. **/
INT32 fOutlink;

/** The PROTECTION field. **/
INT8 fProtection[2];

/** The DETECTORS field. **/
INT8 fDetectors[51];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetPhasingPlanRecord;
```

9.2.15 TNetTimingPlanRecord

This structure is used for network timing plan table records.

```
typedef struct
{
/** The PLAN field. **/
INT32 fPlan;

/** The PHASE field. **/
INT32 fPhase;

/** The NEXTPHASES field. **/
INT8 fNextphases[21];

/** The GREENMIN field. **/
REAL64 fGreenmin;

/** The GREENMAX field. **/
REAL64 fGreenmax;

/** The GREENEXT field. **/
REAL64 fGreenext;

/** The YELLOW field. **/
REAL64 fYellow;

/** The REDCLEAR field. **/
REAL64 fRedclear;
```

```
/** The GROUPFIRST field. **/
INT32 fGroupfirst;

/** The NOTES field. **/
INT8 fNotes[256];

} TNetTimingPlanRecord;
```

9.2.16 TNetDetectorRecord

This structure is used for network detector table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The OFFSET field. **/
REAL64 fOffset;

/** The LANEBEGIN field. **/
INT32 fLanebegin;

/** The LANEEND field. **/
INT32 fLaneend;

/** The LENGTH field. **/
REAL64 fLength;

/** The STYLE field. **/
INT8 fStyle[11];

/** The COORDINATR field. **/
INT8 fCoordinatr[51];

/** The CATEGORY field. **/
INT8 fCategory[11];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetDetectorRecord;
```

9.2.17 TNetSignalCoordinatorRecord

This structure is used for network signal coordinator table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The TYPE field. **/
INT8 fType[11];

/** The ALGORITHM field. **/
INT8 fAlgorithm[11];

/** The NOTES field. **/
INT8 fNotes;

} TNetSignalCoordinatorRecord;
```

9.2.18 TNetActivityLocationRecord

This structure is used for activity location table records.

```
/** Maximum allowed optional user-defined fields in activity
 *  location data.
 **/
#define ACTIVITY_MAX_USER 20

typedef struct
{
/** The ID field. **/
INT32 fId;

/** The NODE field. **/
INT32 fNode;

/** The LINK field. **/
INT32 fLink;

/** The OFFSET field. **/
REAL64 fOffset;

/** The LAYER field. **/
INT8 fLayer[11];

/** The EASTING field. **/
REAL64 fEasting;

/** The NORTHING field. **/
REAL64 fNorthing;
```

```
/** The ELEVATION field. **/
REAL64  fElevation;

/** The number of values in the fUserName and fUser Data arrays.
INT32 fNumberUser;

/** Optional array of user-defined real values.  The number of
 *  values in the array is variable, but must be the same in each
 *  record. The data will typically be related to land use.
 *  The optional fields immediately precede the NOTES field.
 **/
REAL64 fUserData[ACTIVITY_MAX_USER];

/** The names of the fields in fUser Data. **/
INT8 fUserNames[ACTIVITY_MAX_USER] [32];

/** The NOTES field. **/
INT8 fNotes[256];

} TNetActivityLocationRecord;
```

9.2.19 TNetProcessLinkRecord

This structure is used for process link table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The FROMID field. **/
INT32 fFromid;

/** The FROMTYPE field. **/
INT8 fFromtype[11];

/** The TOID field. **/
INT32 fToid;

/** The TOTYPE field. **/
INT8  fTotype[11];

/** The DELAY field. **/
REAL64 fDelay;

/** The COST field. **/
REAL64 fCost;

/** The NOTES field. **/
INT8 fNotes[256];
```

```
} TNetProcessLinkRecord;
```

9.2.20 TNetStudyAreaLinkRecord

This structure is used for network study area link table records.

```
typedef struct
{
/** The ID field. **/
INT32 fId;

/** The BUFFER field. **/
INT8  fBuffer[2];

/** The NOTES field. **/
INT8 fNotes;

} TNetStudyAreaLinkRecord;
```

10. INDEXING

TRANSIMS data files (particularly the activity, plan, output, and iteration database files) may be very large. Furthermore, the following common operations on these files must be efficient:

- modify small, randomly scattered records
- merge modifications back into the original file
- sort on several different keys
- retrieve specified records

File indexing provides a mechanism for efficient use of these large files.

TRANSIMS provides a C library that supports accessing files through an associated index. It also incorporates a particular strategy for using this library within the TRANSIMS framework. This section describes the indexes, library routines, and the way they are used within TRANSIMS.

10.1 Interface Functions

10.1.1 BTree_Create

Signature
```
void BTree_Create(BTree* tree,
const char* data_file,
const char* index_file)
```

Description Creates a new index; does not add any entries to the index file.

Argument `tree` – tree to create; assumes `tree` is a valid pointer.
`date_file` – name of file where the data resides.
`index_file` – name of index file to create.

Return Value None.

10.1.2 BTree_Open

Signature
```
void BTree_Open(BTree* tree, const char* index_file)
```

Description Opens an existing btree index file.

Argument `tree` – tree to open; assumes `tree` is a valid pointer.
`index_file` – name of index file to open.

Return Value None.

10.1.3 BTree_Close

Signature `void **BTree_Close**(BTree* tree)`

Description Closes a btree and releases resources.

Argument `tree` – tree to close; the pointer is not freed.

Return Value None.

10.1.4 BTree_CreateFromFile

Signature
```
BTree* BTree_CreateFromFile(const char* data_file,
const char* index_file, enum act_keys key1,
enum act_keys key2)
```

Description Creates a btree from a given data file.

Argument
`data_file` – datafile from which to read entries.
`index_file` – index file to which entries will be added.
`key1` – field number of primary key.
`key2` – field number of secondary key.

Return Value A new index containing the entries from the data file.

10.1.5 AddFileToIndex

Signature `void **BTree_AddFileToIndex**(BTree* tree, char* data_file)`

Description Adds entries in file to tree.

Argument
`tree` – tree to which entries will be added.
`data_file` – data file from which to take entries.

Return Value None.

10.1.6 BTree_Insert

Signature `void **BTree_Insert**(BTree* tree, BTreeEntry* entry)`

Description Inserts an entry into a btree.

Argument `tree` – index to which entries will be added.

`entry` – the entry to add.

Return Value None.

10.1.7 BTree_AddFilename

Signature `int` **`BTree_AddFilename`**`(BTree* tree, char* filename)`

Description Adds an additional data filename.

Argument `tree` – tree to which filename will be added.
`filename` – date file to add.

Return Value The file number of the added filename.

10.1.8 BTree_GetFilename

Signature `char*` **`BTree_GetFilename`**`(BTree* tree, int i)`

Description Converts from file number in a `BTreeEntry` to file name.

Argument `tree` – tree in which to do the lookup.
`i`– file number to look up.

Return Value The filename of the corresponding data file, or `NULL` if there is no such data file.

10.1.9 BTree_GetFileNumber

Signature
```
int BTree_GetFileNumber(BTree* tree,
const char* filename)
```

Description Converts from file name to file number.

Argument `tree` – tree in which to do the lookup.
`filename` – data file name to look up.

Return Value The filenumber of the corresponding data file, or –1 if there is no such data file.

10.1.10 BTree_ClearFilename

Signature `void` **`BTree_ClearFilename`**`(BTree* tree)`

Description Removes all filenames.

Argument `tree` – tree from which to remove filenames.

Return Value None.

10.1.11 BTree_RenumberFiles

Signature `void BTree_RenumberedFiles(BTree* tree, int dest, int src)`

Description Renumbers filenumber in entries of a tree.

Argument `tree` – tree in which to do the renumbering.
`dest` – the new file number.
`src` – the old file number, if –1 renumber all entries.

Return Value None.

10.1.12 BTree_GetDataPointer

Signature `char* BTree_GetDataPointer(BTree* tree, BTreeEntry* e)`

Description Gets entry in the data file for entry.

Argument `tree` – tree in which to do lookup.
`e` – entry for which to find data.

Return Value A pointer into the mmaped file, or `NULL` if the data is not found. The pointer is not null-terminating (‘\0’). Any changes made through this pointer will be reflected in the data file. This pointer should not be freed.

10.1.13 BTree_GetDataLine

Signature `char* BTree_GetDataLine(BTree* tree, BTreeEntry* e)`

Description Gets entry in the data file for entry.

Argument `tree` – tree in which to do lookup.
`e` – entry for which to find data.

Return Value A copy of the data, or `NULL` if the data is not found. The pointer is null-terminated (‘\0’). Any changes made through this pointer will not be reflected in the data file. The caller is responsible for freeing this pointer.

10.1.14 BTree_FindEntry

Signature `BTreeEntry* BTree_FindEntry(BTree* tree, BTreeEntry* e)`

Description Finds an entry in a tree.

Argument `tree` – the tree in which to do the search.
`e` – entry to find, only needs keys to be set up correctly.

Return Value The complete entry in the tree, or `NULL` if the entry was not found.

10.1.15 BTree_Validate

Signature `void BTree_Validate(BTree* tree, const char* from)`

Description Validates a tree. Currently, checks for the following:
- Proper order of elements in tree
- Correct number of entries
- Stuff in valid subtree
  - valid key types
  - valid file number
  - valid child pointers

Argument `tree` – tree to validate.
`from` – where called from, used to print message (only if problem found).

Return Value None.

10.1.16 BTreeDeleteEntry

Signature `void BTree_DeleteEntry(BTree* tree, BTreeEntry* e)`

Description Deletes an index entry in a tree. Does not modify any data files.

Argument `tree` – tree from which to delete.
`e` – entry to delete.

Return Value None.

10.1.17 BTreeIt_Create

Signature `BTreeIt* BTreeIt_Create(BTree* tree)`

Description Creates an iterator to a tree.

Argument `tree` – the tree into which to point.

Return Value An iterator into the tree. This iterator should be destroyed with **`BTreeIt_Destroy()`** to free all resources. This iterator is invalid if the tree is modified.

10.1.18 BTreeIt-Reset

Signature `void` **`BTreeIt_Reset`**`(BTreeIt* it)`

Description Resets an iterator to point to the first entry of the tree.

Argument `it` – the iterator to reset.

Return Value None.

10.1.19 BTreeIt_Advance

Signature `void` **`BTreeIt_Advance`**`(BTreeIt* it)`

Description Advances the iterator to the next entry in the tree.

Argument `it` – the iterator to advance.

Return Value None.

10.1.20 BTreeIt_MoreData

Signature `int` **`BTreeIt_MoreData`**`(BTreeIt* it)`

Description Are we at the end of the index?

Argument `it` – the iterator to check.

Return Value 0 if there are no more entries; non-zero if there are more entries.

10.1.21 BTreeIt_Get

Signature `BTreeEntry*` **`BTreeIt_Get`**`(BTreeIt* it)`

Description Gets the entry to which the iterator points.

Argument `it` – the iterator to query.

Return Value A pointer to the current entry in the tree, or `NULL` if the iterator is invalid. The entry should not be modified or freed.

10.1.22 BTreeIt_Destroy

Signature `void BTreeIt_Destroy(BTreeIt* it)`

Description Destroys an iterator and frees all resources.

Argument `it` – the iterator to destroy.

Return Value None.

10.1.23 BTreeIt_GetIterator

Signature `BTreeIt* BTreeIt_GetIterator(BTree* tree, BTreeEntry* e)`

Description Returns an iterator pointing to an entry in the tree.

Argument `tree` – tree in which to find the iterator.
`e` – entry to set the iterator to, only needs keys to be set up correctly.

Return Value An iterator that points to `e`; or `NULL` if `e` was not found.

10.1.24 BTreeIt_Compare_Equal

Signature `int BTreeIt_Compare_Equal(BTreeIt* i1, BTreeIt* i2)`

Description Compares two iterators.

Argument `i1, i2` – iterators to compare.

Return Value 0 if the iterators do not point to the same entry in the tree; non-zero if they do point to the same entry.

10.2 Data Structures

10.2.1 Key

This structure is used to represent the value of a key.

```
typedef union u_key
{
/** A key can be either an integer or a floating point number.
**/
```

```
int I;
float f;

} Key;
```

10.2.2 BTreeEntry

This structure is used as an index entry; it holds two keys—the file number and offset where the data resides.

```
typedef struct btree_entry_s
{
/** Primary Key. **/
Key key1;

/** Secondary Key. **/
Key key2;

/** Number of bytes from beginning of file. **/
off_t offset;

/** Number of data file. **/
short file;

/** Key data types. **/
char key_type;

/** Unused. **/
char pad;

} BTreeEntry;
```

10.2.3 BTreeNode

This structure is used as the node of a btree; it holds up to `BTREE_ORDER` entries and `BTREE_order+1` children.

```
typedef struct btree_node_s
{
/** Number of keys currently in this node. **/
int keys;

/** Is this a leaf node? **/
int leaf;

/** Data to be stored. **/
struct btree_entry key [BTREE_ORDER];

/** Child pointers. **/
off_t child[BTREE_ORDER+1];
```

```
/** Padding to make node even multiple of page size. **/
char pad[20];

} BTreeNode;
```

10.2.4 BTree

This structure contains information about a btree. It is sized so that it takes up the first page of the btree index file (`BTREE_PAGESIZE` bytes). One btree can have up to 255 data files, with a combined filename length of 5596 bytes.

```
typedef struct btree_s
{
/** Index of Root of tree. **/
off_t root;

/** Index file. **/
int index_fd;

/** Start of node array. **/
struct btree_node* index;

/** Number of nodes used. **/
size_t size;
/** Number of nodes allocated. **/
size_t allocated;

/** Number of entries in the tree. **/
size_t entries;

/** Height of the tree. **/
size_t height;

/** Field number of key1. **/
short key1;

/** Field number of key2. **/
short key2;

/** Order of this btree, used as sanity check. **/
short order;

/** Number of data files. **/
char num_filenames;

/** Version of btree file, used as sanity check. **/
char version;

/** File Descriptors for data files. **/
int data fd[256];
```

```
/** Pointers to mmaped files. **/
char* data[256];

/** Offset in filename array of filenames. **/
short filename_off[256];

/** Names of index files. **/
char filename[5596];

} BTree;
```

10.2.5 BtreeIt

This structure holds a pointer into a btree index.

```
typedef struct btree_it
{
/** Tree into which this iterator points. **/
BTree* tree;

/** Height of the tree. **/
int height;

/** Level in the tree of the iterator. **/
int level;

/** Path from root of tree to current position. **/
off_t* node;

/** Current key number at each level in path. **/
size_t* key;

} BTreeIt;
```

10.3 Utility Programs

10.3.1 IndexFilenames

The purpose of this tool is to allow easy inspection and reassignment of the data file names referred to by an index.

Each index file maintains a directory listing the names of the data files to which its entries refer, and a default UNIX directory path that is prepended to any filenames that do not begin with the character "/". The directory entries themselves contain pointers into this list of filenames. When a data file is moved, it is more efficient to update the list of filenames than to recreate the index.

This tool can be invoked in either "write" or "read" mode. In write mode, it simply prints the default directory and file names, one per line, into a file. In read mode, it reads the

default directory and file names from a file and overwrites the current settings in the index file.

Usage:

```
IndexFilenames <index> <command> <file>
```

Where `<index>` is the index file to read or modify, `<command>` is "w" to write the names of the data files to `<file>` or "r" to read the names of the data files from `<file>`. The first line of `<file>` is the default directory, which will be prepended to any data file name that does not begin with a "/" or ".". For example, to change the name of location of the data files for the local activities household index, the following commands would be needed:

```
IndexFilenames local.act.hh.idx w names
vi names # edit names of data files
IndexFilename  local.act.hh.idx r names
```

Example:

This example shows how to update the index *plans.tim.idx* if the data files it refers to are moved from */tmp* to */home/eubank*.

```
gershwin 1> $TRANSIMS_HOME/bin/IndexFilenames plans.tim.idx w
names
gershwin 2> cat names
/tmp
plans.1
plans.2
gershwin 3> cat > newnames
/home/eubank
plans.1
plans.2
gershwin 4> $TRANSIMS_HOME/bin/IndexFilenames plans.tim.idx r
newnames
```

Troubleshooting:

It is an error to reduce the number of filenames held in an index's directory, since some entries will no longer point to a valid file name. It is not an error to have duplicate file names, although it may cause inefficient memory use when the index is used.

10.3.2 IndexActivityFile, IndexPlanFile, IndexVehFile

These standalone utilities create index files with the suffixes indicated in **Table 4** in the same directory as the original file.

Usage:

```
IndexVehFile <vehicle_file>
IndexActivityFile <activity_files>
IndexPlanFile <plan_file>
```

If, for example, *plan_file.trv.idx* already exists but no *plan_file.tim.idx* exists, the effect of the last command will be to use that index to create *plan_file.tim.idx*. If neither *plan_file_index* exists, both will be created using *plan_file*.

10.3.3 MergeIndices

The purpose of the *MergeIndices* tool is to merge and update potentially large data files without touching all the data on disk. For example, a 100 Megabyte plan file can be merged with another 100 Megabyte plan file and the result sorted by both departure time and traveler ID simply by merging and sorting the indexes for each file properly.

For each input index specified on the command line, copy the desired entries from that index into an output index. Only those entries whose primary key has not been seen in a previously processed index are desired. The input indexes are processed from last to first, so this restriction essentially means that entries from indexes specified later on the command line overwrite those specified earlier on the command line.

Usage:

```
MergeIndices <output-name> <index1> [<index2> [<index3> ... ]]
```

Example:

The following command will merge the indexes for transit driver plans stored in the file *plans.transit*, plans from the first iteration of the Route Planner stored in *plans.pop.1*, and plans from the second iteration of the Route Planner stored in *plans.pop.2*:

```
MergeIndices out.trv.idx plans.transit.trv.idx plans.pop.1.trv.idx plans.pop.2.trv.idx
```

The output index will be *out.trv.idx*. Assuming all of the transit driver IDs are distinct from other members of the population, *out.trv.idx* will contain all of the transit driver plans, all of the plans from *plans.pop.2*, and plans for all of the travelers in *plans.pop.1* who did not appear in *plans.pop.2*.

The resulting index can be used to create an index sorted by time using the *IndexPlanFile* tool. (Remove any existing *out.tim.idx* first.) Alternatively, the Traffic Microsimulator will create the index sorted by time when it is next run. These indexes can be used directly by the Traffic Microsimulator (or distributed using the *DistributePlans* tool, or viewed using the *PlanFilter* tool) without the need to create an actual file *out* containing all the data for the plan legs. If desired, such a file could be created using the *-d* option of the *PlanFilter* tool.

Troubleshooting:

Only the primary key is used to distinguish entries. Thus, *MergeIndices* works well for plans indexed by traveler ID, but not for plans indexed by departure time. Similarly, if the household ID is used as a key, all travelers in a household should be updated at once.

10.3.4 IndexDefrag

The *IndexDefrag* utility defragments and merges the data files for an index. The entries in an index are written to a new datafile in the order that they appear in the index. The index is modified to use the new data file. For example, if *vehicles.hh.idx* refers to *vehicles1*, and *vehicles2*, then the command

```
IndexDefrag vehicles.hh.idx vehicles.new
```

will create a new datafile, with the entries from *vehicles1* and *vehicles2* that occur in *vehicles.hh.idx*. The index file *vehicles.hh.idx* will now refer only to file *vehicles.new*.

10.4 Files

Table 3: Indexing library files.

Type	File Name	Description
Binary Files	*libTIO.a*	TRANSIMS Interfaces library
Source Files	*btree.h*	Defines `Btree` and `BTreeEntry` data structures and interface functions
	btree.c	*Btree.h* interface functions source file
	btree_it.h	Defines BtreeIt data structure and interface functions
	btree_it.c	*btree_it.h* interface functions source file

10.5 Usage

An index must be created for each file to be accessed by index. Any TRANSIMS component that requires an index file will create it if it does not already exist. Thus, components listed under “Users” in **Table 4**create the corresponding index files, as well as the components and utilities listed under “Creators”. Some TRANSIMS components also automatically index some of the files they create. Index files may also be created by standalone utility programs.

Table 4: Indexes used by TRANSIMS components.

Data File Type	Extension	Major, Minor Sort Keys	Creator(s)	User(s)
Activity file	.hh.idx	Household ID, Person ID	Activity Generator, IndexActivityFile	Route Planner, Iteration Database
Activity file	.trv.idx	Person ID, Household ID	Activity Generator, IndexActivityFile	Traffic microsimulator, Iteration Database
Plan file	.trv.idx	Traveler ID, Activation Time	Route Planner, IndexPlanFile	Traffic Microsimulator, Iteration Database
Plan file	.tim.idx	Activation Time, Traveler ID	PlanFilter, IndexPlanFile	Traffic Microsimulator
Event Output	.trv.idx	Traveler ID, Trip ID		Iteration Database

Data File Type	Extension	Major, Minor Sort Keys	Creator(s)	User(s)
Event Output	.loc.idx	Location ID, Traveler ID		Iteration Database
Vehicle file	.veh.idx	Vehicle ID, Household ID	Population Synthesizer, IndexVehicle File	Activity Generator Route Planner, Traffic Microsimulator
Vehicle file	.hh.idx	Household ID, Vehicle ID	IndexVehicleFile	Activity Generator

Creating an index involves reading each data record in the file, determining the values of the fields to be used as keys, noting the byte offset for the beginning of that record, and inserting an entry into the index (`BTree`). Each index is given a name derived by adding an extension to the base data file. The extension indicates the major sort key for the index and that the file is an index. For example, *.trv.idx* indicates that the file is an index whose major sort key is traveler ID. These extensions are defined in the IO library header files.

Indexes are sorted according to the fields used for the major and minor sort keys. If a data file must be accessed in a particular order, for example by traveler ID, it is more efficient to build an index with that field as the major sort key than to create another data file that has been sorted. Thus, the framework will often expect several different indexes for each data file.

TRANSIMS provides C library routines for creating the indexes used by the framework, as well as standalone utility programs. Given the name of a data file to index, these routines first determine whether the required index files already exist, with a modification date more recent than that of the data file. If so, nothing is done. Where possible, these routines also create an index by examining other available indexes instead of scanning the entire data file. For example, there are two indexes for plan files—one has traveler ID as a major sort key and departure time as the minor key; the other has the sort keys reversed. Thus, one index can be created from the other without looking at the original data.

The user has access to functions used to compare keys. The current functions compare the primary sort key first. If these are equal, they compare the secondary sort keys. It is possible to specify a *don't care* value for the secondary sort key, which will compare equal to any secondary sort key value.

Indexes may be merged. In this case, entries appearing later in the set of indexes replace earlier entries. None of the data in the original data files needs to be moved to merge the indexes, yet iterating through the merged index will yield the same results as if the data files themselves had been merged and sorted.

Similarly, removing entries from an index makes the corresponding data invisible to users accessing the data file through the index.

After several merge, sort, and filter operations, it becomes difficult to determine the contents of the resulting "notional" file except by using the indexing scheme. To support

users who may wish to use other data processing tools, TRANSIMS provides the ability to *defragment* the data pointed to by an index. That is, it provides executables that will create a new file on disk identical to the notional file.

10.6 Examples

```
#include "IO/btree.h"
#include "IO/btree_it.h"

int main(int argc, char* argv[])
{
  char* data_file;
  char* index_file;
  BTreeEntry entry;
  BTree* tree;
  BTreeIt *it;

  index_file = "sample0.idx";
  data_file = "sample1.dat";

  /* Create an index file */
  tree = BTree_CreateFromFile(data_file,
                              index_file,
                              kActivityPerson,
                              kActivityStartMin);

  /* Add a second data file to the index */
  data_file = "sample2.dat";
  BTree_AddFileToIndex(tree, data_file);

  /* Delete an entry */
  entry.key1.i = 0;
  entry.key2.f = 0.0;
  entry.key_type = K_IF;
  BTree_DeleteEntry(tree, &entry);

  /* Use an iterator to examine each entry */
  it = BTreeIt_Create(tree);
  BTreeIt_Reset(it);
  while (BTreeIt_MoreData(it))
  {
    BTreeEntry* e;
    BTreeEntry* e2;
    BTreeIt* it2;
    /* Get the entry for this iterator */
    e = BTreeIt_Get(it);

    /* Get a second iterator, pointing to the same entry */
    it2 = BTreeIt_GetIterator(tree, e);
    /* Get the entry for this iterator */
```

```
    e2 = BTreeIt_Get(it2);
    /* Verify that the entries are the same (they should be) */
    if (!BTree_Compare_Equal(e, e2) || !BTreeIt_Compare_Equal(it,
it2))
    {
      if (!BTree_Compare_Equal(e, e2))
        printf("Entries differ\n");
      if (!BTreeIt_Compare_Equal(it, it2))
        printf("Iterators differ\n  ");
    }
    /* Clean up the second iterator */
    BTreeIt_Destroy(it2);

    /* Advance to the next entry */
    BTreeIt_Advance(it);
  }
  BTreeIt_Destroy(it);

  BTree_Close(tree);
  free(tree);
  return 0;
}
```

11. CONFIGURATION

This section describes the format of configuration files. These files contain the parameters used by the various TRANSIMS software modules.

Configuration files are text files that contain lines of the following types:

- A key followed (optionally) by a value and (optionally) by a comment starting with the pound (#) symbol. The key and the value must be separated by space and/or tab characters.
- A comment line staring with the pound symbol (#).
- A blank line.

11.1 Interface Functions

Functions are available for reading and writing records of a configuration file.

11.1.1 ConfigRead

Signature `int` **`ConfigRead`**`(FILE* file, TConfigRecord* record)`

Description Read a record from a configuration file.

Argument `file` – `FILE` pointer for the configuration file.
`record` – pointer to `TConfigRecord` structure into which the record is read.

Return Value Nonzero if the record was successfully read, or zero if not.

11.1.2 ConfigWrite

Signature
```
int ConfigWrite(FILE* file,
const TConfigRecord* record)
```

Description Write a record to a configuration file.

Argument `file` – `FILE` pointer for the configuration file.
`record` – pointer to TConfigRecord structure from which the record is written.

Return Value Nonzero if the record was successfully written, or zero if not.

11.2 Data Structures

11.2.1 TConfigRecord Structure

Structure for configuration file records.

```
typedef struct
{
/**  The key, if the record has one. **/
INT8 fKey[64];

/**  The value, if the record has one. **/
INT8 fValue[256];

/**  The comment, if the record has one. **/
INT8 fComment[512];

} TConfigRecord;
```

11.3 Utility Programs

11.3.1 SetEnv

The *SetEnv* program takes the keys in a configuration file and converts them into UNIX shell environment variables set to the values corresponding to the keys. Its first argument is the name of the UNIX shell, and its second argument is the name of the configuration file; it does not recurse nested configuration files. It is typically used as follows:

```
eval `SetEnv csh default.config`
eval `SetEnv csh my-run.config`
```

where `default.config` is the default configuration file identified in the configuration file `my-run.config`.

11.4 Files

Table 5: Configuration library files.

Type	File Name	Description
Binary Files	*libTIO.a*	TRANSIMS Interfaces library
Utilities	*SetEnv*	Environment variable setting utilities
Source Files	*configio.h*	Defines configuration file data structures and interface functions
	configio.c	Configuration file interface functions source file

11.5 Configuration File Keys

The configuration file key `CONFIG_DEFAULT_FILE` specifies the name of a configuration file whose keys and values are to be used in cases where a key is not set in the current configuration file.

11.6 Examples

Figure 1 and Figure 2 give examples of typical configuration and default configuration files, respectively. Note that when keys are duplicated in these files, the value in the non-default file takes precedence.

Figure 1: Example configuration file.

```
CONFIG_DEFAULT_FILE /home/transims/allstr-run/default.config

NET_PROCESS_LINK_TABLE Process_Link.minimal.tbl

ROUTER_MAX_DEGREE 15

CA_BIN /home/projects/transims/config/integration/bin/ARCH.PVM.SUN4SOL2/CA
CA_SIM_STEPS 7200
CA_MASTER_MESSAGE_LEVEL 1

PAR_COMMUNICATION   PVM
PAR_SLAVES 1
```

Figure 2: Example default configuration file.

```
###################### GLOBAL PARAMETERS ######################

# The width of a lane in meters
# float
GBL_LANE_WIDTH 3.5

# The length of a cell in meters
# float
GBL_CELL_LENGTH 7.5


###################### NETWORK PARAMETERS #####################

NET_DIRECTORY                /home/transims/allstr-run/network/

NET_NODE_TABLE               Node.tbl
NET_LINK_TABLE               Link.tbl
NET_POCKET_LANE_TABLE        Pocket_Lane.tbl
NET_LANE_USE_TABLE           Lane_Use.tbl
NET_SPEED_TABLE              Speed.tbl
NET_LANE_CONNECTIVITY_TABLE  Lane_Connectivity.tbl
NET_TURN_PROHIBITION_TABLE   Turn_Prohibition.tbl
NET_UNSIGNALIZED_NODE_TABLE  Unsignalized_Node.tbl
NET_SIGNALIZED_NODE_TABLE    Signalized_Node.tbl
NET_PHASING_PLAN_TABLE       Phasing_Plan.tbl
```

```
NET_TIMING_PLAN_TABLE        Timing_Plan.tbl
NET_SIGNAL_COORDINATOR_TABLE Signal_Coordinator.tbl
NET_DETECTOR_TABLE           Detector.tbl
NET_BARRIER_TABLE            Barrier.tbl
NET_PARKING_TABLE            Parking.tbl
NET_TRANSIT_STOP_TABLE       Transit_Stop.tbl
NET_ACTIVITY_LOCATION_TABLE  Activity_Location.tbl
NET_PROCESS_LINK_TABLE       Process_Link.tbl
NET_STUDY_AREA_LINKS_TABLE   Study_Area_Link.tbl


###################### SYNTHETIC POPULATION PARAMETERS ####################

POP_NUMBER_HH           1000
POP_BASELINE_FILE       /home/transims/allstr-run/output/allstr.basepop
POP_LOCATED_FILE        /home/transims/allstr-run/output/allstr.locpop
POP_STARTING_VEHICLE_ID 100000
POP_STARTING_HH_ID      1
POP_STARTING_PERSON_ID  101


###################### ACTIVITY GENERATOR PARAMETERS ####################

ACT_FULL_OUTPUT          /home/transims/allstr-run/output/allstr.activities
ACT_PARTIAL_OUTPUT       /home/transims/allstr-run/output/allstr.partact
ACT_FEEDBACK_FILE        /home/transims/allstr-run/output/allstr.actfeed
ACT_WORK_LOC_ALPHA       1
ACT_WORK_LOC_BETA        1
ACT_WORK_LOC_GAMMA       1
ACT_TIME_ALPHA           1
ACT_TIME_BETA            1
ACT_MODE_ALPHA           1
ACT_MODE_BETA            1
ACT_WORK_LOCATION_OPTION 1
ACT_MODE_CHOICE_OPTION   4
ACT_HOME_HEADER          HOME
ACT_WORK_HEADER          WORK
ACT_ACCESS_HEADER        ACCESS


###################### OUTPUT PARAMETERS ####################

OUT_DIRECTORY               /home/transims/allstr-run/output

OUT_SNAPSHOT_NAME_1         allstr.snapshot
OUT_SNAPSHOT_BEGIN_TIME_1   0
OUT_SNAPSHOT_END_TIME_1     86400
OUT_SNAPSHOT_TIME_STEP_1    1
OUT_SNAPSHOT_EASTING_MIN_1  1
OUT_SNAPSHOT_EASTING_MAX_1  1000000
OUT_SNAPSHOT_NORTHING_MIN_1 1
OUT_SNAPSHOT_NORTHING_MAX_1 1000000
OUT_SNAPSHOT_NODES_1        /home/transims/allstr-run/data/allstr.nodes
OUT_SNAPSHOT_LINKS_1        /home/transims/allstr-run/data/allstr.links
OUT_SNAPSHOT_SUPPRESS_1
OUT_SNAPSHOT_FILTER_1

OUT_EVENT_NAME_1            allstr.event
OUT_EVENT_BEGIN_TIME_1      0
OUT_EVENT_END_TIME_1        86400
OUT_EVENT_TIME_STEP_1       1
OUT_EVENT_EASTING_MIN_1     1
OUT_EVENT_EASTING_MAX_1     1000000
OUT_EVENT_NORTHING_MIN_1    1
OUT_EVENT_NORTHING_MAX_1    1000000
OUT_EVENT_NODES_1           /home/transims/allstr-run/data/allstr.nodes
OUT_EVENT_LINKS_1           /home/transims/allstr-run/data/allstr.links
OUT_EVENT_SUPPRESS_1
OUT_EVENT_FILTER_1

OUT_SUMMARY_NAME_1          allstr.summary
```

```
OUT_SUMMARY_BEGIN_TIME_1    0
OUT_SUMMARY_END_TIME_1      86400
OUT_SUMMARY_TIME_STEP_1     900
OUT_SUMMARY_SAMPLE_TIME_1   60
OUT_SUMMARY_BOX_LENGTH_1    150
OUT_SUMMARY_EASTING_MIN_1   1
OUT_SUMMARY_EASTING_MAX_1   1000000
OUT_SUMMARY_NORTHING_MIN_1  1
OUT_SUMMARY_NORTHING_MAX_1  1000000
OUT_SUMMARY_NODES_1         /home/transims/allstr-run/data/allstr.nodes
OUT_SUMMARY_LINKS_1         /home/transims/allstr-run/data/allstr.links
OUT_SUMMARY_SUPPRESS_1
OUT_SUMMARY_FILTER_1

##################### SIMULATION PARAMETERS #################

# see IO/log.h for possible levels
CA_SLAVE_MESSAGE_LEVEL    0
CA_MASTER_MESSAGE_LEVEL   0

# name of executable (used by Msim.pl)
CA_BIN CA

# the max number of occupants of a bus
# int > 1
CA_BUS_CAPACITY   50

# the number of cells a bus occupies in a jam
# float > 0.0
CA_BUS_LENGTH     2.0

# the acceleration of a car, bus, etc.
# (in cells per timestep per timestep)
# float > 0.0
CA_MAXIMUM_ACCELERATION     0.4
CA_BUS_MAXIMUM_ACCELERATION    0.1

# the maximum speed of a car, bus, etc.
# (in cells per timestep)
# float > 0.0
CA_MAXIMUM_SPEED     4.5
CA_BUS_MAXIMUM_SPEED    2.5

# If nonzero, no attempt will be made to read in transit vehicles
# and transit passengers will not be simulated.
# int(?)
CA_NO_TRANSIT       1

# Some time after a vehicle becomes off plan, it will exit the simulation.
# the probability that a vehicle with speed >= 1 will decelerate by 1
# (also an increment added to the speed limit on a link)
# in the discrete version (not compiled with -DCONTINUOUS)
# float > 0 and < 1
CA_DECELERATION_PROBABILITY   0.2

# use to compute the number of cells that must be vacant in an acceptable gap
# (acceptable gap is speed of oncoming vehicle * Velocity Factor)
# float (> 1.0 ? )
CA_GAP_VELOCITY_FACTOR      3.0

# Probability of proceeding when interfering gap is not acceptable
# in cases of links with competing stop/yield signs
# float > 0 and < 1
CA_IGNORE_GAP_PROBABILITY      0.66

# The number of vehicles which can be buffered in each
# of an intersection's queues (One queue for each lane of each incoming link)
# int > 1
CA_INTERSECTION_CAPACITY     10

# Vehicles take at least this many timesteps to traverse an intersection
```

```
# int >= 0
CA_INTERSECTION_WAIT_TIME     1

# Can't change lanes if random variable drawn on each timestep for each vehicle
# is less than this
# float > 0 and < 1
CA_LANE_CHANGE_PROBABILITY     0.99

# number of cells ahead to look for deciding which lane is best upon entering a link
# int >= 0
CA_LOOK_AHEAD_CELLS       35

# If vehicle has not moved for this many timesteps,
# it becomes off-plan and chooses a different destination link, if possible.
# int >= 0
CA_MAX_WAITING_SECONDS       600

# The exit time is the minimum of the expected arrival time at the destination
# and the current time + OFF_PLAN_EXIT_TIME
# int >= 0
CA_OFF_PLAN_EXIT_TIME   1

# Determines, in a complicated way, whether lane changes for the
# sake of following a plan need to be considered
# int >= 0
CA_PLAN_FOLLOWING_CELLS   70

# specify start time for simulation
# int
CA_SIM_START_HOUR  0
CA_SIM_START_MINUTE  0
CA_SIM_START_SECOND  0

# number of timesteps to simulate
# int >= 0
CA_SIM_STEPS  3600

# send map of locations of all accessories to all slaves
CA_BROADCAST_ACC_CPN_MAP       0

# migrate travelers by broadcasting them
CA_BROADCAST_TRAVELERS         1

# number of time-steps to be executed before slaves synchronize with master
CA_SEQUENCE_LENGTH      1

# Initialize the random seed
# seed48 is called with a pointer to the first element of an array
# of these 3 unsigned shorts
# unsigned short
CA_RANDOM_SEED1  1
CA_RANDOM_SEED2  2
CA_RANDOM_SEED3  3

# Use the cached binary representation of the network database
# in the file specified by CA_NETWORK_FILE
# int
CA_USE_NETWORK_CACHE  0
# string
# CA_NETWORK_FILE

# The following delays model just the time it takes to walk up the steps or
# through the doors or whatever. They have nothing to do with the
# time spent waiting in the queue.

# The mean number of seconds it takes a traveler to board a transit vehicle.
# float >= 0.0
CA_ENTER_TRANSIT_DELAY   1.6

# The mean number of seconds it takes to disembark.
# float >= 0.0
```

```
CA_EXIT_TRANSIT_DELAY  1.8

# The number of seconds after a vehicle reaches the stop before
# passengers can start boarding
CA_TRANSIT_INITIAL_WAIT  5

# Name of a file containing TRANSIMS format vehicle information
# (locations, type, etc.)
CA_VEHICLE_FILE  /home/transims/allstr-run/output/allstr.vehicles

CA_USE_PARTITIONED_ROUTE_FILES  0

CA_LATE_BOUNDARY_RECEPTION     1
CA_PARALLEL_LOG                0

CA_PARALLEL_IO_TEST_MODE       0
CA_PARALLEL_IO_TEST_INTERVAL   30

CA_OUTPUT_BUFFER_COUNT         32

CA_RTM_SAMPLE_INTERVAL         0


###################### TRANSIT PARAMETERS ####################

# Name of a file containing TRANSIMS format transit route information
# (list of stops for each route)
# string
TRANSIT_ROUTE_FILE /home/transims/allstr-run/data/allstr.routes

# Name of a file containing TRANSIMS format transit schedule information
# (list of arrival time for each vehicle at each stop)
# string
TRANSIT_SCHEDULE_FILE  /home/transims/allstr-run/data/allstr.schedules

###################### PLAN PARAMETERS #######################

# Name of a file containing TRANSIMS format legs
# string
PLAN_FILE  /home/transims/allstr-run/output/allstr.plans

###################### ROUTER PARAMETERS #####################

ROUTER_OUTPUT_PLAN_FILE /home/transims/allstr-run/output/allstr.plans
ROUTER_ACTIVITY_FILE    /home/transims/allstr-run/output/allstr.activities
ROUTER_VEHICLE_FILE     /home/transims/allstr-run/output/allstr.vehicles
ROUTER_MODE_MAP_FILE    /home/transims/allstr-run/data/allstr.modes

ROUTER_MAXNFASIZE         5
ROUTER_MAX_DEGREE         15
ROUTER_INTERNAL_PLAN_SIZE 400
ROUTER_VERBOSE  2

# If length < corr_thresh * dist, adjust the length
# float
ROUTER_CORR  0.0

# ??
# float
ROUTER_OVERDO  3.0

# Backdating time of travel information ??
# int
ROUTER_ZERO_BACKD  0

###################### LOGGING PARAMETERS ####################

LOG_LOG_CONFIG        0
LOG_LOAD_NETWORK      1
LOG_PARTITIONING      1
LOG_DISTRIBUTION      1
```

```
LOG_RUNTIMEMONITOR    0
LOG_CONTROL           0
LOG_TIMING            1
LOG_BOUNDARIES        0
LOG_ROUTING           1
LOG_ROUTING_DETAIL    1
LOG_TIMESTEP          1
LOG_TIMESTEP_DETAIL   1
LOG_PARALLEL          0
LOG_VEHICLES          1
LOG_MIGRATION         1
LOG_MIGRATION_DETAIL  1
LOG_TRANSIT           1
LOG_EMISSIONS         1
LOG_IO_DETAIL         0

###################### VISUALIZER PARAMETERS #################

# int, will be single buffered if non-zero
VIS_SINGLE_BUFFERED  0

# Name of a file containing batch commands (unused)
# string
VIS_BATCH_FILE

# The length of a box in meters
# float
VIS_BOX_LENGTH          150.0

###################### PARTITIONING PARAMETERS ###############

PAR_PVM_ROOT                    /sw/Cvol/pvm3
PAR_PVM_ARCH                    SUN4SOL2
PAR_PVM_WAIT_FOR_DEAMON         20

PAR_MPI_ROOT                    /sw/Cvol/mpich
PAR_MPI_ARCH                    solaris
PAR_MPI_DEVICE                  ch_p4

PAR_MIN_CELLS_TO_SPLIT         10
PAR_SLAVES                      2

# if 1, use orthogonal bisection to distribute the network
# otherwise, use the METIS graph partitioning library
# int
PAR_USE_METIS_PARTITION         1
PAR_USE_OB_PARTITION            0

PAR_PARTITION_FILE              /tmp/partition
PAR_SAVE_PARTITION              0

# if 0 use (number of lanes) for edge weight, (length * number of lanes) for edge penalty
#       and 0 for node weights in the partitioning algorithm
# otherwise, use the file named in RTM_FEEDBACK_FILE and RTM_PENALTY_FACTOR.
# int
PAR_USE_RTM_FEEDBACK            0

# Filename for edge and node weights for partitioning
# File format is lines of the form:
# 0 ID Weight
# 1 ID Weight Penalty
# The first line sets a node weight
# the second line sets an edge weight: if penalty is -1, use current value *
RTM_PENALTY_FACTOR
#                                       otherwise use Penalty * RTM_PENALTY_FACTOR
# string
PAR_RTM_FEEDBACK_FILE           /tmp/rtm

# See above for RTM_FEEDBACK_FILE
# float > 0.0
PAR_RTM_PENALTY_FACTOR         100.0
```

```
PAR_REPORT_OUTGOING_LINK_TIME_ONLY    1

##################### SELECTOR PARAMETERS ###################

# Only travelers whose (actual - expected) / expected
# is greater than this will be affected by any operations
# float > 0
SEL_FRUSTRATION_THRESH 1.5

# Fraction of travelers to select for
# just rerouting
# reassigning activities
# choosing a new mode preference
# changing the time of activities
# float, >= 0 and <= 1
SEL_REROUTE_FRAC 0.1
SEL_REASSIGN_FRAC 0.1
SEL_REMODE_FRAC 0.1
SEL_RETIME_FRAC 0.1

# Name of files in which to place traveler ids
# selected for each of the possible changes
# string
SEL_REROUTE_FILE
SEL_REMODE_FILE
SEL_RETIME_FILE
SEL_REASSIGN_FILE

# ================================================================================
# Local Variables:
# tab-width:4
# End:
# ================================================================================
```

12. LOGGING

The TRANSIMS logging interface is to be used for the logging output of all applications that will be part of the TRANSIMS suite of software modules and will be integrated into the development environment. Using a single interface allows the standardization of logging messages.

12.1 Interface Functions

Each logging message is associated with a module passed in the parameter `theSubSystem`. There are predefined modules for most subsystems in TRANSIMS (see *IO/log.h for a list*.)

There are four different message levels that are passed in the parameter `theMessageLevel`:

1) `MSG_PRINT` – a normal informative message. It does NOT describe a warning or an error.
2) `MSG_WARNING` – a warning that may need user attention, but is most likely not to corrupt the application results.
3) `MSG_SEVERE_WARNING` – a warning that does not require the user to shut down the application but will most likely result in corrupted output.
4) `MSG_ERROR` – an actual error message that results in immediate termination of the program.

The parameter `Format` contains the actual message. It is interpreted as a C-style `printf(1)` format string that permits the passing of additional parameters after the format string. There is no need to terminate the format string with a newline character, since that will be automatically added.

Notes:

1) Do not try to by-pass the interface because this may result in messages getting lost.
2) Refrain from using the strings `ERROR` or `WARNING` (or any other pattern listed in the DEFINES-Reserved String Pattern section of the *log.h* file) in your messages. The interface will add appropriate strings to your messages so that they can be identified.
3) Choose the message level with care since “harmless” levels such as `MSG_PRINT` or `MSG_WARNING` may be deactivated when the application is run in production mode. Really important messages should be of type `MSG_SEVERE_WARNING` or `MSG_ERROR`.

4) Do not make any assumption about where the logging output will end up. The default will be standard output, but it may also be redirected to a file.

12.1.1 cMessage

Signature
```
void CMessage(enum TSubsystem theSubSystem,
enum TMessageLevel theMessageLevel,
const char * Format, . . .)
```

Description Write a message using the logging system. See above for more detailed description.

Argument
`theSubSystem` – an enumeration for the subsystem that produces the message.
`theMessageLevel` – one of four levels described above.
`Format` – a string describing the message format.
. . . – any additional parameters that may be needed.

Return Value None.

12.2 Files

Table 6: Logging library files.

Type	File Name	Description
Binary Files	*libTIO.a*	TRANSIMS Interfaces library
Source Files	*log.c*	Source file the logging functions
	log.h	Header file for logging functions

12.3 Examples

```
cMessage (SUB_CA, MSG_WARNING, “More vehicle (%d) than expected (%d)”,
          NrOfVehicle, NrOfExpectedVehicles;
```